UNIVERSITY OF CAMBRIDGE

Baking Pi – Operating Systems Development

This course has not yet been updated to work with the Raspberry Pi models B+ and A+. Some elements may not work, in particular the first few lessons about the LED. It has also not been updated for Raspberry Pi v2.

Welcome to Baking Pi: Operating Systems Development! Course by Alex Chadwick.

> You can now help contribute to this tutorial on GitHub.

This website is here to guide you through the process of developing *very* basic operating systems on the Raspberry Pi! This website is aimed at people aged 16 and upwards, although younger readers may still find some of it accessible, particularly with assistance. More lessons may be added to this course in time.

This course takes you through the basics of operating systems development in assembly code. I have tried not to assume any prior knowledge of operating systems development or assembly code. It may be helpful to have some programming experience, but the course should be accessible without. The Raspberry Pi forums are full of friendly people ready to help you out if you run into

trouble. This course is divided into a series of 'lessons' designed to be taken in order as below. Each 'lesson' includes some theory, and also a practical exercise, complete with a full answer.

Rather than leading the reader through the full details of creating an Operating System, these tutorials focus on achieving a few common tasks separately. Hopefully, by the end, the reader should know enough about Operating Systems that they could try to put together everything they've learned and make one. Although the lessons are generally focused on creating very specific things, there is plenty of room to play with what you learn. Perhaps, after reading the lesson on functions, you imagine a better style of assembly code. Perhaps after the lessons on graphics you imagine a 3D operating system. Since this is an Operating Systems course, you will have the power to design things how you like. If you have an idea, try it! Computer Science is still a young subject, and so there is plenty left to discover!

1 Requirements

1.1 Hardware

In order to complete this course you will need a Raspberry Pi with an SD card and power supply. It is helpful, but not necessary, for your Raspberry Pi to be able to be connected to a screen and keyboard.

In addition to the Raspberry Pi used to test and run your operating system code, you also

need a seperate computer running Linux, Microsoft Windows or Mac OS X capable of writing to the type of SD card used by your Raspberry Pi. This other computer is your development and support system.

1.2 Software

In terms of software, you require a GNU compiler toolchain that targets ARMv6 processors. You will install or build a set of tools, called a cross-compiler, on your development system. This cross-compiler converts your source code files into Raspberry Pi-compatible executable files which are placed on the SD card. The SD card is then transferred to the Raspberry Pi where the executable can be tested.

You can find instruction for getting the toolchain on the Downloads Page, along with model answers for all of the exercises.

2 Lessons

	Name	Description
0	Introduction	This introductory lesson does not contain a practical element, but exists to explain the basic concepts of what is an operating system, what is assembly code,

		and other important basics. If you just want to get straight into practicals, it should be safe to skip this lesson.
OK LED Series (Beginner)		
1	OK01	The OK01 lesson contains an explanation about how to get started and teaches how to enable the 'OK' or 'ACT' LED on the Raspberry Pi board near the RCA and USB ports.
2	OK02	The OK02 lesson builds on OK01, by causing the 'OK' or 'ACT' LED to turn on and off repeatedly.
3	OK03	The OK03 lesson builds on OK02 by teaching how to use functions in assembly to make more

		reusable and rereadable code.
4	OK04	The OK04 lesson builds on OK03 by teaching how to use the timer to flash the 'OK' or 'ACT' LED at precise intervals.
5	OK05	The OK05 lesson builds on OK04 using it to flash the SOS morse code pattern (...---...).
Screen Series (Advanced)		
6	Screen01	The Screen01 lesson teaches some basic theory about graphics, and then applies it to display a gradient pattern to the screen or TV.
7	Screen02	The Screen02 lesson builds on Screen01, by teaching how to draw lines and also a small feature on generating

		pseudo random numbers.
8	Screen03	The Screen03 lesson builds on Screen02 by teaching how to draw text to the screen, and introduces the concept of the kernel command line.
9	Screen04	The Screen04 lesson builds on Screen03 by teaching how to manipulate text to display computed values on the screen.
Input Series (Advanced)		
10	Input01	The Input01 lesson teaches some theory about drivers, and linking programs, as well as keyboards. It is then applied to print out input characters to the screen.
11	Input02	The Input02 lesson builds

	on Input01 by teaching how to make a command line interface for an Operating System.

Table 2.1 - Lessons

Spot a mistake? You can help improve this tutorial on GitHub.

Lesson 0 Introduction

This introductory lesson does not contain a practical element, but exists to explain the basic concepts of what is an operating system, what is assembly code and other important basics. If you just want to get straight into practicals, it should be safe to skip this lesson.

1 Operating Systems

> Throughout these tutorials I will put interesting information in boxes like this one.

> Throughout these tutorials I will put information about each command we learn in boxes like this one.

An operating system is just a very complicated program. It has the job of organising other programs on a computer, including sharing the computer's time, memory, hardware and other resources. Some big families of desktop operating systems that you may have heard of include GNU/Linux, Mac OS X and Microsoft Windows. Other devices also need operating systems such as phones, which may use operating systems such as Android, iOS and Windows Phone.[1]

Since the operating system has to interact with the hardware on a computer system, it also has to have specific knowledge of the hardware on a system. To allow operating

systems to be used on a variety of computers, the concept of **drivers** was invented. Drivers are small bits of code that can be added and removed from the operating system in order to allow the operating system to talk to a particular piece of hardware. In this course, we do not cover how to create such removable drivers, and instead focus on making specific ones for the Raspberry Pi.

There are all kinds of different designs of operating systems, and this course can only just scratch the surface. In this course we will mainly focus on getting the operating system to interact with a variety of bits of hardware, as this is often the trickiest bit, and the part for which the least documentation and help exists online.

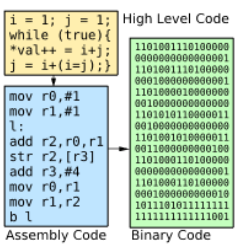
2 Assembly Code

> A processor can often perform millions of instructions per second, but they must be simple.

This course will be written almost exclusively in assembly code. Assembly code is code that is *very* close to what the computer understands. How a computer really works is that there is a small device called a processor which is capable of performing simple jobs like adding numbers, and there is a set of one or more microchips called the **RAM** which are capable of storing numbers. When a computer has power, the processor works through a sequence of instructions given to it by the programmer, which cause it to change numbers in the RAM, and interact with connected hardware. Assembly code is a

translation into human readable text of those commands.

When programming normally, the programmer writes code in a programming language such as C++, Java, C#, Basic, etc, and then a program called the compiler translates what the programmer wrote into assembly code, which is the further reduced into binary code[2]. Binary code is what the computer actually understands, but it is almost impossible for humans to read. Assembly code is much better, but it can be frustrating how few commands are possible. Remember that every command you write in assembly code is something that the processor understands directly, and so the commands are simple by design, as a physical circuit must process each one.



Just like with ordinary programming, there are many different assembly code languages, however unlike ordinary programming, the reason these exist is due to the fact that there exists many different processors, each designed to understand a different language. Thus a program written in assembly code for one machine, will not work on a different one. For most things, this would be a disaster as each program would have to be rewritten for every system it was used on, but for operating systems this isn't so much of a

problem, as it would have to be rewritten anyway due to differing hardware. Nevertheless, most operating systems are written in C++ or C, so that they can be converted more easily, and only the sections that absolutely have to be written in assembly are.

You're now ready to move on to the first lesson, Lesson 1: OK01

---

[1]^ For a more complete list of Operating Systems see List of operating systems - Wikipedia, the free encyclopaedia

[2]^ I am, of course, simplifying this explanation of ordinary programming, in truth it depends heavily on the language and the machine. For the interested, see Compiler - Wikipedia, the free encyclopedia

Spot a mistake? You can help improve this tutorial on GitHub.

Based on contributions at https://github.com/chadderz121/bakingpi-www.

Lesson 1 OK01

The OK01 lesson contains an explanation about how to get started and teaches how to enable the 'OK' or 'ACT' **LED** on the Raspberry Pi board near the RCA and USB ports. This light was originally labelled OK but has been renamed to ACT on the revision 2 Raspberry Pi boards.

1 Getting Started

I am assuming at this point that you have already visited the Downloads page, and got the necessary GNU Toolchain. Also on the downloads page is a file called OS Template. Please download this and extract its contents to a new directory.

2 The Beginning

> The '.s' file extension is commonly used for all forms of assembly code, it is up to us to remember this is ARMv6.

Now that you have extracted the template, create a new file in the 'source' directory called 'main.s'. This file will contain the code for this operating system. To be explicit, the folder structure should look like:

```
build/
  (empty)
source/
  main.s
```

```
kernel.ld
LICENSE
Makefile
```

Open 'main.s' in a text editor so that we can begin typing assembly code. The Raspberry Pi uses a variety of assembly code called ARMv6, so that is what we'll need to write in.

Copy in these first commands.

```
.section .init
.globl _start
_start:
```

As it happens, none of these actually do anything on the Raspberry Pi, these are all instructions to the assembler. The assembler is the program that will translate between assembly code that we understand, and binary machine code that the Raspberry Pi understands. In Assembly Code, each line is a new command. The first line here tells the Assembler[1] where to put our code. The template I provided causes the code in the section called **.init** to be put at the start of the output. This is important, as we want to make sure we can control which code runs first. If we don't do this, the code in the alphabetically first file name will run first! The **.section** command simply tells the assembler which section to put the code in, from this point until the next **.section** or the end of the file.

> In assembly code, you may skip lines, and put spaces before and after commands to aid readability.

The next two lines are there to stop a warning message and aren't all that important.[2]

3 The First Line

Now we're actually going to code something. In assembly code, the computer simply goes through the code, doing each instruction in order, unless told otherwise. Each instruction starts on a new line.

Copy the following instruction.

```
ldr r0, =0x20200000
```

ldr reg, =val puts the number **val** into the register named **reg**.

That is our first command. It tells the processor to store the number 0x20200000 into the register r0. I shall need to answer two questions here, what is a register, and how is 0x20200000 a number?

A single register can store any integer between 0 and 4,294,967,295 inclusive on the Raspberry Pi, which might seem like a large amount of memory, but it is only 32 binary bits.

A register is a tiny piece of memory in the processor, which is where the processor stores the numbers it is working on right now. There are quite a few of these, many of which have a special meaning, which we will come to later. Importantly there are 13 (named r0,r1,r2,...,r9,r10,r11,r12) which are called General Purpose, and you can use

them for whatever calculations you need to do. Since it's the first, I've used r0 in this example, but I could very well have used any of the others. As long as you're consistent, it doesn't matter.

0x20200000 is indeed a number. However it is written in Hexadecimal notation. To learn more about hexadecimal expand the box below:

Hexadecimal explained

Hexadecimal is an alternate system for writing numbers. You may only be aware of the decimal system for writing numbers in which we have 10 digits: 0,1,2,3,4,5,6,7,8 and 9. Hexadecimal is a system with 16 digits: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e and f.

You may recall being taught how decimal numbers work in terms of place value. We say that the rightmost digits is the 'units' digits, the next one left is the 'tens' digit, the next is the 'hundreds' digit, and so on. What this actually meant is, the number is 100 × the value in the 'hundreds' digit, plus 10 × the value in the 'tens' digit, plus 1 × the value in the units digit.

More mathematically, we can now spot the pattern and say that the rightmost digit is the $10^0 = 1$s digit, the next left

is the $10^1 = 10$s digit, the next is $10^2 = 100$s digit, and so on. We have all agreed on the system that 0 is the lowest digit, 1 is the next and so on. But what if we used a different number instead of 10 in these powers? Hexadecimal is just the system in which we use 16 instead.

The mathematics to the right shows that the number 567 in decimal is equivalent to the number 237 in hexadecimal. Often when we need to be clear about what system we're using to write numbers in we put $_{10}$ for decimal and $_{16}$ for hexadecimal. Since it's difficult to write small numbers in assembly code, we use 0x instead to represent a number in hexadecimal notation. So 0x237 means 237_{16}.

So where do a,b,c,d,e and f come in? Well, in order to be able to write every number in hexadecimal, we need extra digits. For example $9_{16} = 9\times16^0 = 9_{10}$, but $10_{16} = 1\times16^1 + 1\times16^0 = 16_{10}$. So if we just used 0,1,2,3,4,5,6,7,8 and 9 we would not be able to write 10_{10}, 11_{10}, 12_{10}, 13_{10}, 14_{10}, 15_{10}. So we introduce 6 new digits such that $a_{16} = 10_{10}$, $b_{16} = 11_{10}$, $c_{16} = 12_{10}$, $d_{16} = 13_{10}$, $e_{16} = 14_{10}$, $f_{16} = 15_{10}$

So, we now have another system for writing numbers. But why did we bother? Well, it turns out that since computers always work in binary, hexadecimal notation is very useful

because every hexadecimal digit is exactly four binary digits long. This has the nice side effect that a lot of computer numbers are round numbers in hexadecimal, even though they're not in decimal. For example, in the assembly code just above I used the number 20200000_{16}. If I had chose to write this in decimal it would have been 538968064_{10}, which is much less memorable.

To convert numbers from decimal to hexadecimal I find the following method easiest:

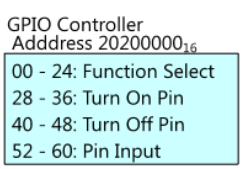
1. Start with the decimal number, say 567.
2. Divide by 16 and calculate the remainder. For example $567 \div 16 = 35$ remainder 7.
3. The remainder is the last digit of the answer in hexadecimal, in the example this is 7.
4. Repeat steps 2 and 3 again with the result of the last division until the result is 0. For example $35 \div 16 = 2$ remainder 3, so 3 is the next digit of the answer. $2 \div 16 = 0$ remainder 2, so 2 is the next digit of the answer.
5. Once the result of the division is 0, you can stop. The answer is just the remainders in the reverse order to which you got them, so $567_{10} = 237_{16}$.

To convert hexadecimal numbers back to decimal, it is easiest to expand out

the number, so $237_{16} = 2\times16^2 + 3\times16^1 + 7\times16^0 = 2\times256 + 3\times16 + 7\times1 = 512 + 48 + 7 = 567$.

So our first command is to put the number 20200000_{16} into r0. That doesn't sound like it would be much use, but it is. In computers, there are an awful lot of chunks of memory and devices. In order to access them all, we give each one an address. Much like a postal address or a web address this is just a means of identifying the location of the device or chunks of memory we want. Addresses in computers are just numbers, and so the number 20200000_{16} happens to be the address of the GPIO controller. This is just a design decision taken by the manufacturers, they could have used any other address (providing it didn't conflict with anything else). I know this address only because I looked it up in a manual[3], there is no particular system to the addresses (other than that they are all large round numbers in hexadecimal).

4 Enabling Output



Having read the manual, I know we're going to need to send two messages to the GPIO controller. We need to talk its language, but if we do, it will obligingly do what we want and turn on the OK LED. Fortunately, it is such a simple chip, that it only needs a few

numbers in order to understand what to do.

```
mov r1,#1
lsl r1,#18
str r1,[r0,#4]
```

mov reg,#val puts the number **val** into the register named **reg**.

lsl reg,#val shifts the binary representation of the number in **reg** by **val** places to the left.

str reg, [dest,#val] stores the number in **reg** at the address given by **dest** + **val**.

These commands enable output to the 16th GPIO pin. First we get a necessary value in r1, then send it to the GPIO controller. Since the first two instructions are just trying to get a value into r1, we could use another **ldr** command as before, but it will be useful to us later to be able to set any given GPIO pin, so it is better to deduce the value from a formula than write it straight in. The OK LED is wired to the 16th GPIO pin, and so we need to send a command to enable the 16th pin.

The value in r1 is needed to enable the LED pin. The first line puts the number 1_{10} into r1. The **mov** command is faster than the **ldr** command, because it does not involve a memory interaction, whereas **ldr** loads the value we want to put into the register from memory. However, **mov** can only be used to load certain values[4]. In ARM assembly code, almost every instruction begins with a

three letter code. This is called the mnemonic, and is supposed to hint at what the operation does. **mov** is short for move and **ldr** is short for load register. **mov** moves the second argument **#1** into the first **r1**. In general, # must be used to denote numbers, but we have already seen a counterexample to this.

The second instruction is **lsl** or logical shift left. This means shift the binary representation for the first argument left by the second argument. In this case this will shift the binary representation of 1_{10} (which is 1_2) left by 18 places (making it $100000000000000000_2 = 262144_{10}$).

If you are unfamiliar with binary, expand the box below:

Binary explained

Just like hexadecimal binary is another way of writing numbers. In binary we only have 2 digits, 0 and 1. This is useful for computers because we can implement this in a circuit by saying that electricity flowing through the circuit means 1, and not means 0. This is how computers actually work and do maths. Despite only having 2 digits binary can still be used to represent every number, it just takes a lot longer.

The image shows the binary representation of the number 567_{10} which is 1000110111_2. We use $_2$ to

denote numbers written in binary.

One of the quirks of binary that we make heavy use of in assembly code is the ease by which numbers can be multiplied or divided by powers of 2 (e.g. 1,2,4,8,16). Normally multiplications and divisions are tricky operations, however these special cases are very easy, and so are very important.

Shifting a binary number left by **n** places is the same as multiplying the number by 2**n**. So, if we want to multiply by 4, we just shift the number left 2 places. If we want to multiply by 256 we could shift it left by 8 places. If we wanted to multiply by a number like 12, we could instead multiply it by 8, then separately by 4 and add the results ($N \times 12 = N \times (8 + 4) = N \times 8 + N \times 4$).

Shifting a binary number right by **n** places is the same as dividing the number by 2**n**. The remainder of the division is the bits that were lost when shifted right. Unfortunately dividing by a binary number that is not an exact power of 2 is very difficult, and will be covered in Lesson 9: Screen04.

This diagram shows common

terminology used with binary. A bit is a single binary digit. A nibble is 4 binary bits. A byte is 2 nibbles, or 8 bits. A half is half the size of a word, 2 bytes in this case. A word refers to the size of the registers on a processor, and so on the Raspberry Pi this is 4 bytes. The convention is to number the most significant bit of a word 31, and the least significant bit as 0. The top, or high bits refer to the most significant bits, and the low or bottom bits refer to the least significant. A kilobyte (KB) is 1000 bytes, a megabyte is 1000 KB. There is some confusion as to whether this should be 1000 or 1024 (a round number in binary). As such, the new international standard is that a KB is 1000 bytes, and a Kibibyte (KiB) is 1024 bytes. A Kb is 1000 bits, and a Kib is 1024 bits.

The Raspberry Pi is little endian by default, meaning that loading a byte from an address you just wrote a word to will load the lowest byte of the word.

Once again, I only know that we need this value from reading the manual[3]. The manual says that there is a set of 24 bytes in the GPIO controller, which determine the settings of the GPIO pin. The first 4 relate to the first 10 GPIO pins, the second 4 relate to the next 10 and so on. There are 54 GPIO pins, so we need 6 sets of 4 bytes, which is 24 bytes in total. Within each 4 byte section, every 3 bits relates to a particular GPIO pin. Since we want the 16th GPIO pin, we need

the second set of 4 bytes because we're dealing with pins 10-19, and we need the 6th set of 3 bits, which is where the number 18 (6×3) comes from in the code above.

Finally the **str** 'store register' command stores the value in the first argument, **r1** into the address computed from the expression afterwards. The expression can be a register, in this case **r0**, which we know to be the GPIO controller address, and another value to add to it, in this case **#4**. This means we add 4 to the GPIO controller address and write the value in **r1** to that location. This happens to be the location of the second set of 4 bytes that I mentioned before, and so we send our first message to the GPIO controller, telling it to ready the 16th GPIO pin for output.

5 A Sign Of Life

Now that the LED is ready to turn on, we need to actually turn it on. This means sending a message to the GPIO controller to turn pin 16 off. Yes, *turn it off*. The chip manufacturers decided it made more sense[5] to have the LED turn on when the GPIO pin is off. Hardware engineers often seem to take these sorts of decisions, seemingly just to keep OS Developers on their toes. Consider yourself warned.

```
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
```

Hopefully you should recognise all of the above commands, if not their values. The first puts a 1 into **r1** as before. The second

shifts the binary representation of this 1 left by 16 places. Since we want to turn pin 16 off, we need to have a 1 in the 16th bit of this next message (other values would work for other pins). Finally we write it out to the address which is 40_{10} added to the GPIO controller address, which happens to be the address to write to turn a pin off (28 would turn the pin on).

6 Happily Ever After

It might be tempting to finish now, but unfortunately the processor doesn't know we're done. In actuality, the processor never will stop. As long as it has power, it continues working. Thus, we need to give it a task to do forever more, or the Raspberry Pi will crash (not much of a problem in this example, the light is already on).

```
loop$:
b loop$
```

name: labels the next line **name**.

b label causes the next line to be executed to be **label**.

The first line here is not a command, but a label. It names the next line **loop$**. This means we can now refer to the line by name. This is called a label. Labels get discarded when the code is turned into binary, but they're useful for our benefit for referring to lines by name, not number (address). By convention we use a **$** for labels which are

only important to the code in this block of code, to let others know they're not important to the overall program. The **b** (branch) command causes the next line to be executed to be the one at the label specified, rather than the one after it. Therefore, the next line to be executed will be this **b**, which will cause it to be executed again, and so on forever. Thus the processor is stuck in a nice infinite loop until it is switched off safely.

The new line at the end of the block is intentional. The GNU toolchain expects all assembly code files to end in an empty line, so that it is sure you were really finished, and the file hasn't been cut off. If you don't put one, you get an annoying warning when the assembler runs.

7 Pi Time

So we've written the code, now to get it onto the pi. Open a terminal on your computer and change the current working directory to the parent directory of the source directory. Type **make** and then press enter. If any errors occur, please refer to the troubleshooting section. If not, you will have generated three files. kernel.img is the compiled image of your operating system. kernel.list is a listing of the assembly code you wrote, as it was actually generated. This is useful to check that things were generated correctly in future. The kernel.map file contains a map of where all the labels ended up, which can be useful for chasing around values.

To install your operating system, first of all get a Raspberry PI SD card which has an

operating system installed already. If you browse the files in the SD card, you should see one called kernel.img. Rename this file to something else, such as kernel_linux.img. Then, copy the file kernel.img that **make** generated onto the SD Card. You've just replaced the existing operating system with your own. To switch back, simply delete your kernel.img file, and rename the other one back to kernel.img. I find it is always helpful to keep a backup of you original Raspberry Pi operating system, in case you need it again.

Put the SD card into a Raspberry Pi and turn it on. The OK LED should turn on. If not please see the troubleshooting page. If so, congratulations, you just wrote your first operating system. See Lesson 2: OK02 for a guide to making the LED flash on and off.

---

[1]^ OK, I'm lying it tells the linker, which is another program used to link several assembled files together. It doesn't really matter.

[2]^ Clearly they're important to you. Since the GNU toolchain is mainly used for creating programs, it expects there to be an entry point labelled **_start**. As we're making an operating system, the **_start** is always whatever comes first, which we set up with the **.section .init** command. However, if we don't say where the entry point is, the toolchain gets upset. Thus, the first line says that we are going to define a symbol called **_start** for all to see (globally), and the second line says to make the symbol **_start** the address of the next line. We will come onto addresses shortly.

[3]^^ This tutorial is designed to spare you the pain of reading it, but, if you must, it can be found here SoC-Peripherals.pdf. For added confusion, the manual uses a different addressing system. An address listed as 0x7E200000 would be 0x20200000 in our OS.
[4]^ Only values which have a binary representation which only has 1s in the first 8 bits of the representation. In other words, 8 1s or 0s followed by only 0s.
[5]^ A hardware engineer was kind enough to explain this to me as follows:

The reason is that modern chips are made of a technology called CMOS, which stands for Complementary Metal Oxide Semiconductor. The Complementary part means each signal is connected to two transistors, one made of material called N-type semiconductor which is used to pull it to a low voltage and another made of P-type material to pull it to a high voltage. Only one transistor of the pair turns on at any time, otherwise we'd get a short circuit. P-type isn't as conductive as N-type, which means the P-type transistor has to be about 3 times as big to provide the same current. This is why LEDs are often wired to turn on by pulling them low, because the N-type is stronger at pulling low than the P-type is in pulling high.

There's another reason. Back in the 1970s chips were made out of entirely out of N-type material ('NMOS'), with the P-type replaced by a resistor. That means that when a signal is pulled low the chip is consuming power (and getting hot) even while it isn't doing anything. Your phone

getting hot and flattening the battery when it's in your pocket doing nothing wouldn't be good. So signals were designed to be 'active low' so that they're high when inactive and so don't take any power. Even though we don't use NMOS any more, it's still often quicker to pull a signal low with the N-type than to pull it high with the P-type. Often a signal that's 'active low' is marked with a bar over the top of the name, or written as SIGNAL_n or /SIGNAL. But it can still be confusing, even for hardware engineers!

Spot a mistake? You can help improve this tutorial on GitHub.

Based on contributions at https://github.com/chadderz121/bakingpi-www.

Lesson 2 OK02

The OK02 lesson builds on OK01, by causing the 'OK' or 'ACT' LED to turn on and off repeatedly. It is assumed you have the code for the Lesson 1: OK01 operating system as a basis.

1 Waiting

Waiting is a surprisingly useful part of Operating System development. Often Operating Systems find themselves with nothing to do, and must delay. In this example, we wish to do so in order to allow the LED flashing off and on to be visible. If you just turned it off and on, it would not be visible, as the computer would be able to turn it off and on many thousands of times per second. In later lessons we will look at accurate waiting, but for now it is sufficient to simply waste time.

```
mov r2,#0x3F0000
wait1$:
sub r2,#1
cmp r2,#0
bne wait1$
```

sub reg,#val subtracts the number **val** from the value in **reg**.

cmp reg,#val compares the value in **reg** with the number **val**.

Suffix **ne** causes the command to be executed only if the last

comparison determined that the numbers were not equal.

The code above is a generic piece of code that creates a delay, which thanks to every Raspberry Pi being basically the same, is roughly the same time. How it does this is using a **mov** command to put the value $3F0000_{16}$ into **r2**, and then subtracting 1 from this value until it is 0. The new commands here are **sub**, **cmp**, and **bne**.

sub is the subtract command, and simply subtracts the second argument from the first.

cmp is a more interesting command. It compares the first argument with the second, and remembers the result of the comparison in a special register called the current processor status register. You don't really need to worry about this, suffice to say it remembers, among other things, which of the two numbers was bigger or smaller, or if they were equal.[1]

bne is actually just a branch command in disguise. In the ARM assembly language family, any instruction can be executed conditionally. This means that the instruction is only run if the last comparison had a certain result. We will use this extensively later for interesting tricks, but in this case we use the **ne** suffix on the **b** command to mean 'only branch if the last comparison's result was that the values were not equal'. The **ne** suffix can be used on any command, as can several other (16 in all) conditions such as **eq** for equal and **lt** for less than.

2 The All Together

I mentioned briefly last time that the status LED can be turned off again by writing to an offset of 28 from the GPIO controller instead of 40 (i.e. **str r1, [r0,#28]**). Thus, you need to modify the code from OK01 to turn the LED on, run the wait code, turn it off, run the wait code again, and then include a branch back to the beginning. Note, it is not necessary to re-enable the output to GPIO 16, we need only do that once. If you're being efficient, which I strongly encourage, you should be able to reuse the value of **r1**. As with all lessons, a full solution to this can be found on the download page. Be careful to make sure all of your labels are unique. When you write **wait1$:** you cannot label another line **wait1$**.

On my Raspberry Pi it flashes about twice a second. this could easily be altered by changing the value we set **r2** to. However, unfortunately we can't precisely predict the speed this runs at. If you didn't manage to get this working see our trouble shooting page, otherwise, congratulations.

In this lesson we've learnt two more assembly commands, **sub** and **cmp**, as well as learning about conditional execution in ARM.

In the next lesson, Lesson 3: OK03 we will evaluate how we're coding, and establish some standards so that we can reuse code, and if necessary, work with C or C++ code.

---

[1]^ I suppose if you've followed the link you really do want to know about it. The CPSR is a 32 bit register consisting of many individual bit fields. It has bit fields for

positive, zero and negative. When a cmp instruction is issued, it subtracts the second argument from the first, and notes down whether it is positive, zero or negative with these fields. Zero means the numbers were equal (a-b = 0 implies a = b), positive means a is bigger than b (a-b > 0 implies a > b) and negative less than. A variety of other comparison instructions exist, but cmp is the most intuitive.

Spot a mistake? You can help improve this tutorial on GitHub.

Based on contributions at https://github.com/chadderz121/bakingpi-www.

Lesson 3 OK03

The OK03 lesson builds on OK02 by teaching how to use functions in assembly to make more reusable and rereadable code. It is assumed you have the code for the Lesson 2: OK02 operating system as a basis.

1 Reusable Code

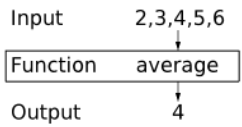
So far we've made code for our operating system by typing the things we want to happen in order. This is fine for such tiny programs, but if we wrote the whole system like this, the code would be completely unreadable. Instead we use functions.

Functions explained

A function is a piece of code that can be reused to compute a certain kind of answer, or perform a certain action. You may also hear them called procedures, routines or subroutines. Although these are all different, people rarely use the correct term.

You should already be happy with the concept of a function from mathematics. For example the cosine function applied to a number gives another number between -1 and 1 which is the cosine of the angle. Notationally we write cos(x) to be the cosine function applied to the value x.

In code, functions can take multiple inputs (including none), give multiple outputs (including none), and may cause side effects. For example a function might create a file on the file system, named after the first input, with length based on the second.



> Functions are said to be 'black boxes'. We put inputs in, and outputs come out, but we don't need to know how they work.

In higher level code such as C or C++, functions are part of the language itself. In assembly code, functions are just ideas we have.

Ideally we want to be able to set our registers to some input values, branch to an address, and expect that at some point the code will branch back to our code having set the registers to output values. This is what a function is in assembly code. The difficulty comes in what system we use for setting the registers. If we just used any system we felt like, each programmer may use a different system, and would find other programmers' work hard to understand. Further, compilers would not be able to work with assembly code as easily, as they would not know how to use the functions. To prevent confusion, a standard called the Application Binary Interface (ABI) was devised for each assembly language which is an agreement on how functions should be run. If everyone

makes functions in the same way, then everyone will be able to use each others' functions. I will teach that standard here, and from now on I will code all of my functions to meet the standard.

The standard says that r0,r1,r2 and r3 will be used as inputs to a function in order. If a function needs no inputs, then it doesn't matter what value it takes. If it needs only one it always goes in r0, if it needs two, the first goes in r0, and the second goes on r1, and so on. The output will always be in r0. If a function has no output, it doesn't matter what value r0 takes.

Further, it also requires that after a function is run, r4 to r12 must have the same values as they had when the function started. This means that when you call a function, you can be sure the r4 to r12 will not change value, but you cannot be so sure about r0 to r3.

When a function completes it has to branch back to the code that started it. This means it must know the address of the code that started it. To facilitate this, there is a special register called lr (link register) which always holds the address of the instruction after the one that called this function.

Register	Brief	Preserved	Rules
r0	Argument and result	No	r0 and r1 are used for passing the first two arguments to functions, and

			returning the results of functions. If a function does not use them for a return value, they can take any value after a function.
r1	Argument and result	No	
r2	Argument	No	r2 and r3 are used for passing the second two arguments to functions. There values after a function is called can be anything.
r3	Argument	No	
r4	General purpose	Yes	r4 to r12 are used for working values, and their value after a

			function is called must be the same as before.
r5	General purpose	Yes	
r6	General purpose	Yes	
r7	General purpose	Yes	
r8	General purpose	Yes	
r9	General purpose	Yes	
r10	General purpose	Yes	
r11	General purpose	Yes	
r12	General purpose	Yes	
lr	Return address	No	lr is the address to branch back to when a function is finished, but this does have to contain the same address after the function has finished.
sp	Stack pointer	Yes	sp is the stack pointer,

	described below. Its value must be the same after the function has finished.

Table 1.1 ARM ABI register usage

Often functions need to use more registers than just r0 to r3. But, since r4 to r12 must stay the same after the method has run, they must be saved somewhere. We save them on something called the stack.

Stack explained

A stack is a metaphor we use in computing for a method of storing values. Just like in a stack of plates, you can only remove items from the top of a stack, and only add items to the top of the stack.

The stack is a brilliant idea for storing registers on when functions are running. For example if I have a function which needs to use registers r4 and r5, it could place the current values of those registers on a stack. At the end of the method it could take them back off again. What is most clever is that if my function had to run another function in order to complete

and that function needed to save some registers, it could put those on the top of the stack while it ran, and then take them off again at the end. That wouldn't affect the values of r4 and r5 that my method had to save, as they would be added to the top of the stack, and then taken off again.

The terminology we used to refer to the values put on the stack by a particular method is that methods 'stack frame'. Not every method needs a stack frame, some don't need to store values.

Because the stack is so useful, it has been implemented in the ARMv6 instruction set directly. A special register called sp (stack pointer) holds the address of the stack. When items are added to the stack, the sp register updates so that it always holds the address of the first item on the stack. **push {r4,r5}** would put the values in r4 and r5 onto the top of the stack and **pop {r4,r5}** would take them back off again (in the correct order).

2 Our First Function

Now that we have some idea about how functions work, let's try to make one. For a basic first example, we are going to make a function that takes no input, and gives an output of the GPIO address. In the last lesson, we just wrote in this value, but it would be better as a function, since it is something we might need to do often in a real operating system, and we might not always remember the address.

Copy the following code into a new file called 'gpio.s'. Just make the new file in the 'source' directory with 'main.s'. We're going to put all functions related to the GPIO controller in one file to make them easier to find.

```
.globl GetGpioAddress
GetGpioAddress:
ldr r0, =0x20200000
mov pc,lr
```

.globl lbl makes the label **lbl** accessible from other files.

mov reg1,reg2 copies the value in **reg2** into **reg1**.

This is a very simple complete function. The **.globl GetGpioAddress** command is a message to the assembler to make the label **GetGpioAddress** accessible to all files. This means that in our main.s file we can branch to the label **GetGpioAddress** even though it is not defined in that file.

You should recognise the **ldr r0, =0x20200000** command, which stores the GPIO controller address in r0. Since this is a function, we have to give the output in r0, so we are not as free to use any register as we once were.

mov pc,lr copies the value in **lr** to **pc**. As mentioned earlier **lr** always contains the address of the code that we have to go back to when a method finishes. **pc** is a special register which always contains the address of the next instruction to be run. A normal branch command just changes the value of

this register. By copying the value in **lr** to **pc** we just change the next line to be run to be the one we were told to go back to.

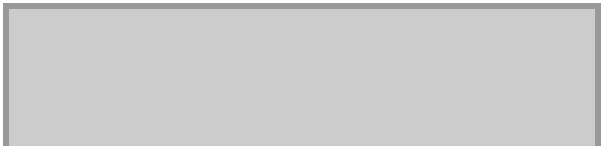
A reasonable question would now be, how would we actually run this code? A special type of branch **bl** does what we need. It branches to a label like a normal branch, but before it does it updates **lr** to contain the address of the line after the branch. That means that when the function finishes, the line it will go back to will be the one after the **bl** command. This makes a function running just look like any other command, it simply runs, does whatever it needs to do, and then carries on to the next line. This is a really useful way of thinking about functions. We treat them as 'black boxes' in that when we use them, we don't need to think about how they work, we just need to know what inputs they need, and what outputs they give.

For now, don't worry about using the function, we will use it in the next section.

3 A Big Function

Now we're going to implement a bigger function. Our first job was to enable output on GPIO pin 16. It would be nice if this was a function. We could simply specify a pin and a function as the input, and the function would set the function of that pin to that value. That way, we could use the code to control any GPIO pin, not just the LED.

Copy the following commands below the GetGpioAddress function in gpio.s.



```
.globl SetGpioFunction
SetGpioFunction:
cmp r0,#53
cmpls r1,#7
movhi pc,lr
```

Suffix **ls** causes the command to be executed only if the last comparison determined that the first number was less than or the same as the second. Unsigned.

Suffix **hi** causes the command to be executed only if the last comparison determined that the first number was higher than the second. Unsigned.

One of the first things we should always think about when writing functions is our inputs. What do we do if they are wrong? In this function, we have one input which is a GPIO pin number, and so must be a number between 0 and 53, since there are 54 pins. Each pin has 8 functions, numbered 0 to 7 and so the function code must be too. We could just assume that the inputs will be correct, but this is very dangerous when working with hardware, as incorrect values could cause very bad side effects. Therefore, in this case, we wish to make sure the inputs are in the right ranges.

To do this we need to check that r0 $<=$ 53 and r1 $<=$ 7. First of all, we can use the comparison we've seen before to compare the value of r0 with 53. The next instruction, **cmpls** is a normal comparison instruction that will only be run if r0 was lower than or the same as 53. If that was the case, it compares r1 with 7, otherwise the result of the comparison is the same as before. Finally

we go back to the code that ran the function if the result of the last comparison was that the register was higher than the number.

The effect of this is exactly what we want. If r0 was bigger than 53, then the **cmpls** command doesn't run, but the **movhi** does. If r0 is < = 53, then the **cmpls** command does run, and so **r1** is compared with 7, and then if it is higher than 7, **movhi** is run, and the function ends, otherwise **movhi** does not run, and we know for sure that r0 < = 53 and r1 < = 7.

There is a subtle difference between the **ls** (lower or same) and **le** (less or equal) as well as between **hi** (higher) and **gt** (greater) suffixes, but I will cover this later.

Copy these commands below the above.

```
push {lr}
mov r2,r0
bl GetGpioAddress
```

push {reg1,reg2,...} copies the registers in the list **reg1,reg2,...** onto the top of the stack. Only general purpose registers and lr can be pushed.

bl lbl sets **lr** to the address of the next instruction and then branches to the label **lbl**.

These next three commands are focused on calling our first method. The **push {lr}** command copies the value in **lr** onto the top of the stack, so that we can retrieve it later. We must do this because when we call GetGpioAddress, we will need to use **lr** to

store the address to come back to in our function.

If we did not know anything about the GetGpioAddress function, we would have to assume it changes r0,r1,r2 and r3, and would have to move our values to r4 and r5 to keep them the same after it finishes. Fortunately, we do know about GetGpioAddress, and we know it only changes r0 to the address, it doesn't affect r1,r2 or r3. Thus, we only have to move the GPIO pin number out of r0 so it doesn't get overwritten, but we know we can safely move it to r2, as GetGpioAddress doesn't change r2.

Finally we use the **bl** instruction to run GetGpioAddress. Normally we use the term 'call' for running a function, and I will from now. As discussed earlier **bl** calls a function by updating the lr to the next instruction's address, and then branching to the function.

When a function ends we say it has 'returned'. When the call to GetGpioAddress returns, we now know that r0 contains the GPIO address, r1 contains the function code and r2 contains the GPIO pin number. I mentioned earlier that the GPIO functions are stored in blocks of 10, so first we need to determine which block of ten our pin number is in. This sounds like a job we would use a division for, but divisions are very slow indeed, so it is better for such small numbers to do repeated subtraction.

Copy the following code below the above.

```
functionLoop$:
    cmp r2,#9
```

```
subhi r2,#10
addhi r0,#4
bhi functionLoop$
```

add reg,#val adds the number **val** to the contents of the register **reg**.

This simple loop code compares the pin number to 9. If it is higher than 9, it subtracts 10 from the pin number, and adds 4 to the GPIO Controller address then runs the check again.

The effect of this is that r2 will now contain a number from 0 to 9 which represents the remainder of dividing the pin number by 10. r0 will now contain the address in the GPIO controller of this pin's function settings. This would be the same as GPIO Controller Address + 4 × (GPIO Pin Number ÷ 10).

Finally, copy the following code below the above.

```
add r2, r2,lsl #1
lsl r1,r2
str r1,[r0]
pop {pc}
```

Argument shift **reg,lsl #val** shifts the binary representation of the number in **reg** left by **val** before using it in the operation before.

lsl reg,amt shifts the binary representation of the number in **reg** left by the number in **amt**.

str reg, [dst] is the same as **str reg, [dst,#0]** .

pop {reg1,reg2,...} copies the values from the top of the stack into the register list **reg1,reg2,...**. Only general purpose registers and pc can be popped.

This code finishes off the method. The first line is actually a multiplication by 3 in disguise. Multiplication is a big and slow instruction in assembly code, as the circuit can take a long time to come up with the answer. It is much faster sometimes to use some instructions which can get the answer quicker. In this case, I know that r2 $\times$ 3 is the same as r2 $\times$ 2 + r2. It is very easy to multiply a register by 2 as this is conveniently the same as shifting the binary representation of the number left by one place.

One of the very useful features of the ARMv6 assembly code language is the ability to shift an argument before using it. In this case, I add **r2** to the result of shifting the binary representation of **r2** to the left by one place. In assembly code, you often use tricks such as this to compute answers more easily, but if you're uncomfortable with this, you could also write something like **mov r3,r2**; **add r2,r3**; **add r2,r3**.

Now we shift the function value left by a number of places equal to r2. Most instructions such as **add** and **sub** have a variant which uses a register rather than a number for the amount. We perform this shift because we want to set the bits that correspond to our pin number, and there are three bits per pin.

We then store the the computed function

value at the address in the GPIO controller. We already worked out the address in the loop, so we don't need to store it at an offset like we did in OK01 and OK02.

Finally, we can return from this method call. Since we pushed **lr** onto the stack, if we pop **pc**, it will copy the value that was in **lr** at the time we pushed it into **pc**. This would be the same as having used **mov pc,lr** and so the function call will return when this line is run.

The very keen may notice that this function doesn't actually work correctly. Although it sets the function of the GPIO pin to the requested value, it causes all the pins in the same block of 10's functions to go back to 0! This would likely be quite annoying in a system which made heavy use of the GPIO pins. I leave it as a challenge to the interested to fix this function so that it does not overwrite other pins values by ensuring that all bits other than the 3 that must be set remain the same. A solution to this can be found on the downloads page for this lesson. Functions that you may find useful are **and** which computes the Boolean and function of two registers, **mvns** which computes the Boolean not and **orr** which computes the Boolean or.

4 Another Function

So, we now have a function which takes care of the GPIO pin function setting. We now need to make a function to turn a GPIO pin on or off. Rather than having one function for off and one function for on, it would be handy to have a single function which does either.

We will make a function called SetGpio which takes a GPIO pin number as its first input in r0, and a value as its second in r1. If the value is 0 we will turn the pin off, and if it is not zero we will turn it on.

Copy and paste the following code at the end of 'gpio.s'.

```
.globl SetGpio
SetGpio:
pinNum .req r0
pinVal .req r1
```

alias .req reg sets **alias** to mean the register **reg**.

Once again we need the **.globl** command and the label to make the function accessible from other files. This time we're going to use register aliases. Register aliases allow us to use a name other than just r0 or r1 for registers. This may not be so important now, but it will prove invaluable when writing big methods later, and you should try to use aliases from now on. **pinNum .req r0** means that **pinNum** now means **r0** when used in instructions.

Copy and paste the following code after the above.

```
cmp pinNum,#53
movhi pc,lr
push {lr}
mov r2,pinNum
.unreq pinNum
pinNum .req r2
bl GetGpioAddress
```

```
gpioAddr .req r0
```

.unreq alias removes the alias **alias**.

Like in SetGpioFunction the first thing we must do is check that we were actually given a valid pin number. We do this in exactly the same way by comparing **pinNum** (**r0**) with 53, and returning immediately if it is higher. Once again we wish to call GetGpioAddress, so we have to preserve **lr** by pushing it onto the stack, and to move **pinNum** to **r2**. We then use the **.unreq** statement to remove our alias from **r0**. Since the pin number is now stored in **r2** we want our alias to reflect this, so we remove the alias from **r0** and remake it on **r2**. You should always **.unreq** every alias as soon as it is done with, so that you cannot make the mistake of using it further down the code when it no longer exists.

We then call GetGpioAddress, and we create an alias for **r0** to reflect this.

Copy and paste the following code after the above.

```
pinBank .req r3
lsr pinBank,pinNum,#5
lsl pinBank,#2
add gpioAddr,pinBank
.unreq pinBank
```

lsr dst,src,#val shifts the binary representation of the number in **src** right by **val**, but stores the result in **dst**.

The GPIO controller has two sets of 4 bytes each for turning pins on and off. The first set

in each case controls the first 32 pins, and the second set controls the remaining 22. In order to determine which set it is in, we need to divide the pin number by 32. Fortunately this is very easy, at is the same as shifting the binary representation of the pin number right by 5 places. Hence, in this case I've named **r3** as **pinBank** and then computed **pinNum** ÷ 32. Since it is a set of 4 bytes, we then need to multiply the result of this by 4. This is the same as shifting the binary representation left by 2 places, which is the command that follows. You may wonder if we could just shift it right by 3 places, as we went right then left. This won't work however, as some of the answer may have been rounded away when we did ÷ 32 which may not be if we just ÷ 8.

The result of this is that **gpioAddr** now contains either 20200000_{16} if the pin number is 0-31, and 20200004_{16} if the pin number is 32-53. This means if we add 28_{10} we get the address for turning the pin on, and if we add 40_{10} we get the address for turning the pin off. Since we are done with **pinBank**, I use **.unreq** immediately afterwards.

Copy and paste the following code after the above.

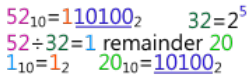
```
and pinNum,#31
setBit .req r3
mov setBit,#1
lsl setBit,pinNum
.unreq pinNum
```

and reg,#val computes the Boolean and function of the

number in **reg** with **val**.

This next part of the function is for generating a number with the correct bit set. For the GPIO controller to turn a pin on or off, we give it a number with a bit set in the place of the remainder of that pin's number divided by 32. For example, to set pin 16, we need a number with the 16th bit a 1. To set pin 45 we would need a number with the 13th bit 1 as $45 \div 32 = 1$ remainder 13.

The **and** command computes the remainder we need. How it does this is that the result of an and operation is a number with 1s in all binary digits which had 1s in both of the inputs, and 0s elsewhere. This is a fundamental binary operation, and is very quick. We have given it inputs of **pinNum** and $31_{10} = 11111_2$. This means that the answer can only have 1 bits in the last 5 places, and so is definitely between 0 and 31. Specifically it only has 1s where there were 1s in **pinNum**'s last 5 places. This is the same as the remainder of a division by 32. It is no coincidence that 31 = 32 - 1.



The rest of this code simply uses this value to shift the number 1 left. This has the effect of creating the binary number we need.

Copy and paste the following code after the above.

```
teq pinVal,#0
.unreq pinVal
streq setBit,[gpioAddr,#40]
```

```
strne setBit,[gpioAddr,#28]
.unreq setBit
.unreq gpioAddr
pop {pc}
```

teq reg,#val checks if the number in **reg** is equal to **val**.

This code ends the method. As stated before, we turn the pin off if pinVal is zero, and on otherwise. **teq** (test equal) is another comparison operation that can only be used to test for equality. It is similar to cmp but it does not work out which number is bigger. If all you wish to do is test if to numbers are the same, you can use **teq**.

If **pinVal** is zero, we store the **setBit** at 40 away from the GPIO address, which we already know turns the pin off. Otherwise we store it at 28, which turns the pin on. Finally, we return by popping the **pc**, which sets it to the value that we stored when we pushed the link register.

5 A New Beginning

Finally, after all that work we have our GPIO functions. We now need to alter 'main.s' to use them. Since 'main.s' is now getting a lot bigger and more complicated, it is better design to split it into two sections. The '.init' we've been using so far is best kept as small as possible. We can change the code to reflect this easily.

Insert the following just after **_start:** in main.s:

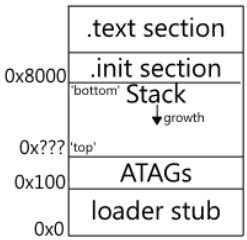
```
```

```
b main

.section .text
main:
mov sp,#0x8000
```

The key change we have made here is to introduce the **.text** section. I have designed the makefile and linker scripts such that code in the .text section (which is the default section) is placed after the .init section which is placed at address 8000_{16}. This is the default load address and gives us some space to store the stack. As the stack exists in memory, it has to have an address. The stack grows down memory, so that each new value is at a lower address, thus making the 'top' of the stack, the lowest address.

> The 'ATAGs' section in the diagram is a place where information about the Raspberry Pi is stored such as how much memory it has, and what its default screen resolution is.



Replace all the code that set the function of the GPIO pin with the following:

```
pinNum .req r0
pinFunc .req r1
mov pinNum,#16
mov pinFunc,#1
bl SetGpioFunction
```

```
.unreq pinNum
.unreq pinFunc
```

This code calls SetGpioFunction with the pin number 16 and the pin function code 1. This has the effect of enabling output to the OK LED.

Replace any code which turns the OK LED on with the following:

```
pinNum .req r0
pinVal .req r1
mov pinNum,#16
mov pinVal,#0
bl SetGpio
.unreq pinNum
.unreq pinVal
```

This code uses SetGpio to turn off GPIO pin 16, thus turning on the OK LED. If we instead used **mov pinVal,#1**, it would turn the LED off. Replace your old code to turn the LED off with that.

6 Onwards

Hopefully now, you should be able to test what you have made on the Raspberry Pi. We've done a large amount of code this time, so there is a lot that can go wrong. If it does, head to the troubleshooting page.

When you get it working, congratulations. Although our operating system does nothing more than it did in Lesson 2: OK02, we've learned a lot about functions and formatting, and we can now code new features much more quickly. It would be very simple now to

make an Operating System that alters any GPIO register, which could be used to control hardware!

In Lesson 4: OK04, we will address our wait function, which is currently imprecise, so that we can gain better control over our LED, and ultimately over all of the GPIO pins.

Spot a mistake? You can help improve this tutorial on GitHub.

Lesson 4 OK04

The OK04 lesson builds on OK03 by teaching how to use the timer to flash the 'OK' or 'ACT' LED at precise intervals. It is assumed you have the code for the Lesson 3: OK03 operating system as a basis.

1 A New Device

> The timer is the only way the Pi can keep time. Most computers have a battery powered clock to keep time when off.

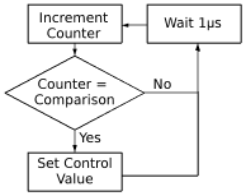
So far, we've only looked at one piece of hardware on the Raspberry Pi, namely the GPIO Controller. I've simply told you what to do, and it happened. Now we're going to look at the timer, and I'm going to lead you through understanding how it works.

Just like the GPIO Controller, the timer has an address. In this case, the timer is based at 20003000_{16}. Reading the manual, we find the following table:

Address	Size / Bytes	Name	Description	Read or Write
20003000	4	Control / Status	Register used to control and clear timer channel comparator matches.	RW

20003004	8	Counter	A counter that increments at 1MHz. R
20003010	4	Compare 0	0th Comparison register. RW
20003014	4	Compare 1	1st Comparison register. RW
20003018	4	Compare 2	2nd Comparison register. RW
2000301C	4	Compare 3	3rd Comparison register. RW

Table 1.1 GPIO Controller Registers



This table tells us a lot, but the descriptions in the manual of the various fields tell us the most. The manual explains that the timer fundamentally just increments the value in Counter by 1 every 1 micro second. Each time it does so, it compares the lowest 32 bits (4 bytes) of the counter's value with the 4 comparison registers, and if it matches any of them, it updates Control / Status to reflect which ones matched.

For more information about bits, bytes, bit fields, and data sizes expand the box below.

Bits explained

A bit is a name for a single binary digit. As you may recall, a single binary digit is either a 1 or a 0.

A byte is the name we give for a collection of 8 bits. Since each bit can be one of two values, there are $2^8 = 256$ different possible values for a byte. We normally interpret a byte as a binary number between 0 and 255 inclusive.

A bit field is another way of interpreting binary. Rather than interpreting it as a number, binary can be interpreted as many different things. A bit field treats binary as a series of switches which are either on (1) or off (0). If we have a meaning for each of these little switches, we can use them to control things. We have actually already met bitfields with the GPIO controller, with the setting a pin on or off. The bit that was a 1 was the GPIO pin to actually turn on or off. Sometimes we need more options than just on or off, so we group several of the switches together, such as with the GPIO controller function settings (pictured), in which every group of 3 bits controls one GPIO pin function.

Our goal is to implement a function that we can call with an amount of time as an input that will wait for that amount of time and then return. Think for a moment about how we could do this, given what we have.

I see there being two options:

1. Read a value from the counter, and then keep branching back into the same code until the counter is the amount of time to wait more than it was.
2. Read a value from the counter, add the amount of time to wait, store this in one of the comparison registers and then keep branching back into the same code until the Control / Status register updates.

> Issues like these are called concurrency problems, and can be almost impossible to fix.

Both of these strategies would work fine, but in this tutorial we will only implement the first. The reason is because the comparison registers are more likely to go wrong, as during the time it takes to add the wait time and store it in the comparison register, the counter may have increased, and so it would not match. This could lead to very long unintentional delays if a 1 micro second wait is requested (or worse, a 0 microsecond wait).

2 Implementation

> Large Operating Systems normally use the Wait function as an opportunity to perform background tasks.

I will largely leave the challenge of creating the ideal wait method to you. I suggest you put all code related to the timer in a file

called 'systemTimer.s' (for hopefully obvious reasons). The complicated part about this method, is that the counter is an 8 byte value, but each register only holds 4 bytes. Thus, the counter value will span two registers.

The following code blocks are examples.

```
ldrd r0,r1,[r2,#4]
```

ldrd regLow,regHigh, [src,#val] loads 8 bytes from the address given by the number in **src** plus **val** into **regLow** and **regHigh** .

An instruction you may find useful is the **ldrd** instruction above. It loads 8 bytes of memory across 2 registers. In this case, the 8 bytes of memory starting at the address in **r2** would be copied into **r0** and **r1**. What is slightly complicated about this arrangement is that **r1** actually holds the highest 4 bytes. In other words, if the counter had a value of $999{,}999{,}999{,}999_{10}$ = $111010001101010010100101000011111111111_2$, **r1** would contain 11101000_2 and **r0** would contain $11010100101001010000111111111111_2$.

The most sensible way to implement this would be to compute the difference between the current counter value and the one from when the method started, and then to compare this with the requested amount of time to wait. Conveniently, unless you wish to support wait times that were 8 bytes, the value in **r1** in the example above could be discarded, and only the low 4 bytes of the counter need be used.

When waiting you should always be sure to use higher comparisons not equality comparisons, as if you try to wait for the gap between the time the method started and the time it ends to be exactly the amount requested, you could miss the value, and wait forever.

If you cannot figure out how to code the wait function, expand the box below for a guide.

Wait function implementation

Borrowing the idea from the GPIO controller, the first function we should write should be to get the address of this system timer. An example of this is shown below:

```
.globl GetSystemTimerBase
GetSystemTimerBase:
ldr r0, =0x20003000
mov pc,lr
```

Another function that will prove useful would be one that returns the current counter value in registers **r0** and **r1**:

```
.globl GetTimeStamp
GetTimeStamp:
push {lr}
bl GetSystemTimerBase
ldrd r0,r1,[r0,#4]
pop {pc}
```

This function simply uses the GetSystemTimerBase function and

loads in the counter value using **ldrd** like we have just learned.

Now we actually want to code our wait method. First of all, we need to know the counter value when the method started, which we can now get using GetTimeStamp.

```
delay .req r2
mov delay,r0
push {lr}
bl GetTimeStamp
start .req r3
mov start,r0
```

This code copies our method's input, the amount of time to delay, into **r2**, and then calls GetTimeStamp, which we know will return the current counter value in **r0** and **r1**. It then copies the lower 4 bytes of the counter's value to **r3**.

Next we need to compute the difference between the current counter value and the reading we just took, and then keep doing so until the gap between them is at least the size of **delay**.

```
loop$:
	bl GetTimeStamp
	elapsed .req r1
	sub elapsed,r0,start
	cmp elapsed,delay
	.unreq elapsed
	bls loop$
```

This code will wait until the requested amount of time has passed. It takes a reading from the counter, subtracts the initial value from this reading and then compares that to the requested delay. If the amount of time that has elapsed is less than the requested delay, it branches back to **loop$**.

```
.unreq delay
.unreq start
pop {pc}
```

This code finishes off the function by returning.

3 Another Blinking Light

Once you have what you believe to be a working wait function, change 'main.s' to use it. Alter everywhere you wait to set the value of r0 to some big number (remember it is in microseconds) and then test it on the Raspberry Pi. If it does not function correctly please see our troubleshooting page.

Once it is working, congratulations you have now mastered another device, and with it, time itself. In the next and final lesson in the OK series, Lesson 5: OK05 we shall use all we have learned to flash out a pattern on the LED.

Spot a mistake? You can help improve this tutorial on GitHub.

[image]

Based on contributions at https://
github.com/chadderz121/bakingpi-www.

Lesson 5 OK05

The OK05 lesson builds on OK04 using it to flash the SOS Morse Code pattern (...---...). It is assumed you have the code for the Lesson 4: OK04 operating system as a basis.

1 Data

So far, all we've had to do with our operating system is provide instructions to be followed. Sometimes however, instructions are only half the story. Our operating systems may need data.

> Some early Operating Systems did only allow certain types of data in certain files, but this was generally found to be too restrictive. The modern way does make programs a lot more complicated however.

In general data is just values that are important. You are probably trained to think of data as being of a specific type, e.g. a text file contains text, an image file contains an image, etc. This is, in truth, just an idea. All data on a computer is just binary numbers, how we choose to interpret them is what counts. In this example we're going to store a light flashing sequence as data.

At the end of 'main.s' copy the following code:

```
.section .data
.align 2
```

```
pattern:
.int
0b11111111101010100010001000101010
```

.align num ensures the address of the next line is a multiple of $2^{\textbf{num}}$.

.int val outputs the number **val** .

To differentiate between data and code, we put all the data in the .data. I've included this on the operating system memory layout diagram here. I've just chosen to put the data after the end of the code. The reason for keeping our data and instructions separate is so that if we eventually implement some security on our operating system, we need to know what parts of the code can be executed, and what parts can't.

I've used two new commands here. **.align** and **.int**. **.align** ensures alignment of the following data to a specified power of 2. In this case I've used **.align 2** which means that this data will definitely be placed at an address which is a multiple of $2^2 = 4$. It is really important to do this, because the **ldr** instruction we used to read memory only works at addresses that are multiples of 4.

The **.int** command copies the constant after it into the output directly. That means that $11111111101010100010001000101010_2$ will be placed into the output, and so the label pattern actually labels this piece of data as pattern.

One challenge with data is finding an efficient and useful

representation. This method of storing the sequence as on and off units of time is easy to run, but would be difficult to edit, as the concept of a Morse - or . is lost.

As I mentioned, data can mean whatever you want. In this case I've encoded the Morse Code SOS sequence, which is ...---... for those unfamiliar. I've used a 0 to represent a unit of time with the LED off, and a 1 to represent a unit of time with the LED on. That way, we can write some code which just displays a sequence in data like this one, and then all we have to do to make it display a different sequence is change the data. This is a very simple example of what operating systems must do all the time; interpret and display data.

Copy the following lines before the **loop$** label in 'main.s'.

```
ptrn .req r4
ldr ptrn,=pattern
ldr ptrn,[ptrn]
seq .req r5
mov seq,#0
```

This code loads the pattern into **r4**, and loads 0 into **r5**. **r5** will be our sequence position, so we can keep track of how much of the pattern we have displayed.

The following code puts a non-zero into **r1** if and only if there is a 1 in the current part of the pattern.

```
mov r1,#1
lsl r1,seq
and r1,ptrn
```

This code is useful for your calls to SetGpio, which must have a non-zero value to turn the LED off, and a value of zero to turn the LED on.

Now modify all of your code in 'main.s' so that each loop the code sets the LED based on the current sequence number, waits for 250000 micro seconds (or any other appropriate delay), and then increments the sequence number. When the sequence number reaches 32, it needs to go back to 0. See if you can implement this, and for an extra challenge, try to do it using only 1 instruction (solution in the download).

2 Time Flies When You're Having Fun...

You're now ready to test this on the Raspberry Pi. It should flash out a sequence of 3 short pulses, 3 long pulses and then 3 more short pulses. After a delay, the pattern should repeat. If it doesn't work please see our troubleshooting page.

Once it works, congratulations you have reached the end of the OK series of tutorials.

In this series we've learnt about assembly code, the GPIO controller and the System Timer. We've learnt about functions and the ABI, as well as several basic Operating System concepts, and also about data.

You're now ready to move onto one of the more advanced series.

- The Screen series is next and teaches you how to use the screen with

assembly code.
- The Input series teaches you how to use the keyboard and mouse.

By now you already have enough information to make Operating Systems that interact with the GPIO in other ways. If you have any robot kits, you may want to try writing a robot operating system controlled with the GPIO pins!

Spot a mistake? You can help improve this tutorial on GitHub.

Lesson 6 Screen01

Welcome to the Screen lesson series. In this series, you will learn how to control the screen using the Raspberry Pi in assembly code, starting at just displaying random data, then moving up to displaying a fixed image, displaying text and then formatting numbers into text. It is assumed that you have already completed the OK series, and so things covered in this series will not be repeated here.

This first screen lesson teaches some basic theory about graphics, and then applies it to display a gradient pattern to the screen or TV.

1 Getting Started

It is expected that you have completed the OK series, and so functions in the 'gpio.s' file and 'systemTimer.s' file from that series will be called. If you do not have these files, or prefer to use a correct implementation, download the solution to OK05.s. The 'main.s' file from here will also be useful, up to and including **mov sp,#0x8000**. Please delete anything after that line.

2 Computer Graphics

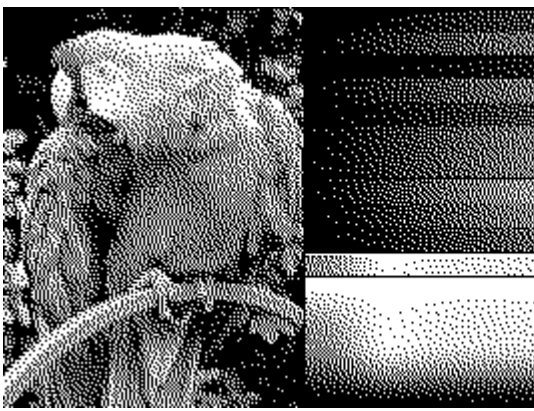
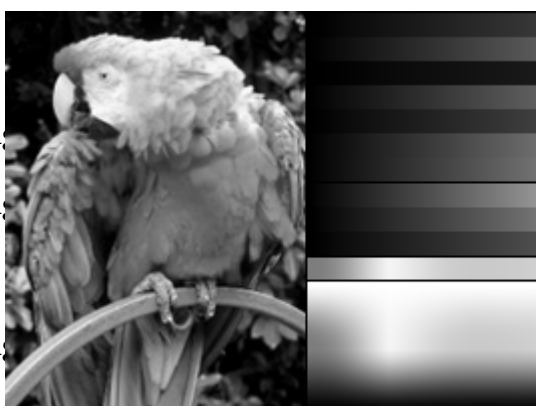
There are a few systems for representing colours as numbers. Here we focus on RGB systems, but HSL is another common system used.

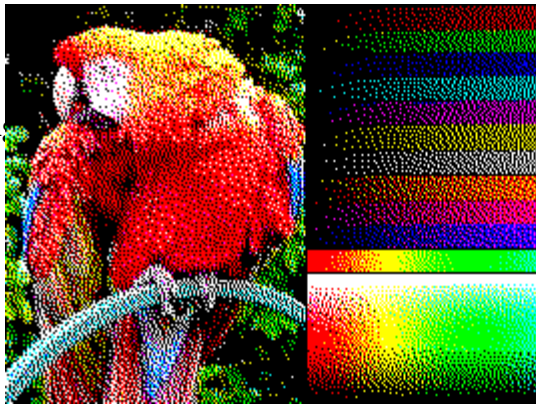
As you're hopefully beginning to appreciate, at a fundamental level, computers are very stupid. They have a limited number of instructions, almost exclusively to do with maths, and yet somehow they are capable of doing many things. The thing we currently wish to understand is how a computer could possibly put an image on the screen. How would we translate this problem into binary? The answer is relatively straightforward; we devise some system of numbering each colour, and then we store one number for every pixel on the screen. A pixel is a small dot on your screen. If you move very close, you will probably be able to make out individual pixels on your screen, and be able to see that everything image is just made out of these pixels in combination.

As the computer age advanced, people wanted more and more complicated graphics, and so the concept of a graphics card was invented. The graphics card is a secondary processor on your computer which only exists to draw images to the screen. It has the job of turning the pixel value information into light intensity levels to be transmitted to the screen. On modern computers, graphics cards can also do a lot more than that, such as drawing 3D graphics. In this tutorial however, we will just concentrate on the first use of graphics cards; getting pixel colours from memory out to the screen.

One issue that is raised immediately by all this is the system we use for numbering colours. There are several choices, each producing outputs of different quality. I will outline a few here for completeness.

Although some images here have few colours they use a technique called spatial dithering. This allows them to still show a good representation of the image, with very few colours. Many early Operating Systems used this technique.

Name	Unique Colours	Description	Examples
Monochrome	2	Use 1 bit to store each pixel, with a 1 being white, and a 0 being black.	
Greyscale	256	Use 1 byte to store each pixel, with 255 representing white, 0 representing black, and all values in between representing a linear combination of the two.	
8 Colour	8	Use 3 bits to store each pixel, the first bit representing the presence of	

		a red channel, the second representin a green channel and the third a blue channel. 
Low Colour	256	Use 8 bits to store each pixel, the first 3 bit representin the intensity of the red channel, the next 3 bits representing the intensity of the green channel and the final 2 bits representing the intensity of the blue channel. 
High Colour	65,536	Use 16 bits to store each pixel, the first 5 bit

		representing the intensity of the red channel, the next 6 bits representing the intensity of the green channel and the final 5 bits representing the intensity of the blue channel.
True Colour	16,777,216	Use 24 bits to store each pixel, the first 8 bits representing the intensity of the red channel, the second 8 representing the green channel and the final 8 bits the blue channel.
RGBA32	16,777,216 with 256	Use 32 bits to store

transparency levels

each pixel, the first 8 bits representing the intensity of the red channel, the second 8 representing the green channel, the third 8 bits the blue channel, and the final 8 bits a transparency channel. The transparency channel is only considered when drawing one image on top of another and is stored such that a value of 0 indicates the image behind's colour, a

	value of 255 represents this image's colour, and all values between represent a mix.

Table 2.1 Some Colour Palettes

In this tutorial we shall use High Colour initially. As you can see form the image, it is produces clear, good quality images, but it doesn't take up as much space as True Colour. That said, for quite a small display of 800x600 pixels, it would still take just under 1 megabyte of space. It also has the advantage that the size is a multiple of a power of 2, which greatly reduces the complexity of getting information compared with True Colour.

> Storing the frame buffer places a heavy memory burden on a computer. For this reason, early computers often cheated, by, for example, storing a screens worth of text, and just drawing each letter to the screen every time it is refreshed separately.

The Raspberry Pi has a very special and rather odd relationship with it's graphics processor. On the Raspberry Pi, the graphics processor actually runs first, and is responsible for starting up the main processor. This is very unusual. Ultimately it doesn't make too much difference, but in many interactions, it often feels like the processor is secondary, and the graphics processor is the most important. The two

communicate on the Raspberry Pi by what is called the 'mailbox'. Each can deposit mail for the other, which will be collected at some future point and then dealt with. We shall use the mailbox to ask the graphics processor for an address. The address will be a location to which we can write the pixel colour information for the screen, called a frame buffer, and the graphics card will regularly check this location, and update the pixels on the screen appropriately.

3 Programming the Postman

> Message passing is quite a common way for components to communicate. Some Operating Systems use virtual message passing to allow programs to communicate.

The first thing we are going to need to program is a 'postman'. This is just two methods: MailboxRead, reading one message from the mailbox channel in r0. and MailboxWrite, writing the value in the top 28 bits of r0 to the mailbox channel in r1. The Raspberry Pi has 7 mailbox channels for communication with the graphics processor, only the first of which is useful to us, as it is for negotiating the frame buffer.

The following table and diagrams describe the operation of the mailbox.

Address	Size / Bytes	Name	Description	Read / Write
2000B880	4	Read	Receiving mail.	R
2000B890	4	Poll	Receive	R

		without retrieving.	
2000B894	Sender	Sender information.	R
2000B898	Status	Information.	R
2000B89C	Configuration	Settings.	RW
2000B8A0	Write	Sending mail.	W

Table 3.1 Mailbox Addresses

In order to send a message to a particular mailbox:

1. The sender waits until the Status field has a 0 in the top bit.
2. The sender writes to Write such that the lowest 4 bits are the mailbox to write to, and the upper 28 bits are the message to write.

In order to read a message:

1. The receiver waits until the Status field has a 0 in the 30th bit.
2. The receiver reads from Read.
3. The receiver confirms the message is for the correct mailbox, and tries again if not.

If you're feeling particularly confident, you now have enough information to write the two methods we need. If not, read on.

As always the first method I recommend you implement is one to get the address of the mailbox region.

```
.globl GetMailboxBase
GetMailboxBase:
ldr r0, = 0x2000B880
mov pc,lr
```

The sending procedure is least complicated, so we shall implement this first. As your methods become more and more complicated, you will need to start planning them in advance. A good way to do this might be to write out a simple list of the steps that need to be done, in a fair amount of detail, like below.

1. Our input will be what to write (r0), and what mailbox to write it to (r1). We must validate this is by checking it is a real mailbox, and that the low 4 bits of the value are 0. Never forget to validate inputs.
2. Use GetMailboxBase to retrieve the address.
3. Read from the Status field.
4. Check the top bit is 0. If not, go back to 3.
5. Combine the value to write and the channel.
6. Write to the Write.

Let's handle each of these in order.

1.
```
.globl MailboxWrite
MailboxWrite:
tst r0,#0b1111
movne pc,lr
cmp r1,#15
movhi pc,lr
```

> **tst reg,#val** computes **and reg,#val** and compares the result with 0.

This achieves our validation on **r0** and **r1**. **tst** is a function that compares two numbers by computing the logical and

operation of the numbers, and then comparing the result with 0. In this case it checks that the lowest 4 bits of the input in r0 are all 0.

2.
```
channel .req r1
value .req r2
mov value,r0
push {lr}
bl GetMailboxBase
mailbox .req r0
```

This code ensures we will not overwrite our value, or link register and calls GetMailboxBase.

3.
```
wait1$:
status .req r3
ldr status,[mailbox,#0x18]
```

This code loads in the current status.

4.
```
tst status,#0x80000000
.unreq status
bne wait1$
```

This code checks that the top bit of the status field is 0, and loops back to 3. if it is not.

5.
```
add value,channel
.unreq channel
```

This code combines the channel and value together.

```
```

6.
```
str value,[mailbox,#0x20]
.unreq value
.unreq mailbox
pop {pc}
```

   This code stores the result to the write field.

The code for MailboxRead is quite similar.

1. Our input will be what mailbox to read from (r0). We must validate this is by checking it is a real mailbox. Never forget to validate inputs.
2. Use GetMailboxBase to retrieve the address.
3. Read from the Status field.
4. Check the 30th bit is 0. If not, go back to 3.
5. Read from the Read field.
6. Check the mailbox is the one we want, if not go back to 3.
7. Return the result.

Let's handle each of these in order.

1.
```
.globl MailboxRead
MailboxRead:
cmp r0,#15
movhi pc,lr
```

   This achieves our validation on **r0**.

2.
```
channel .req r1
mov channel,r0
push {lr}
bl GetMailboxBase
mailbox .req r0
```

This code ensures we will not overwrite our value, or link register and calls GetMailboxBase.

3.
```
rightmail$:
wait2$:
status .req r2
ldr status,[mailbox,#0x18]
```

This code loads in the current status.

4.
```
tst status,#0x40000000
.unreq status
bne wait2$
```

This code checks that the 30th bit of the status field is 0, and loops back to 3. if it is not.

5.
```
mail .req r2
ldr mail,[mailbox,#0]
```

This code reads the next item from the mailbox.

6.
```
inchan .req r3
and inchan,mail,#0b1111
teq inchan,channel
.unreq inchan
bne rightmail$
.unreq mailbox
.unreq channel
```

This code checks that the channel of the mail we just read is the one we were supplied. If not it loops back to 3.

```
```

7.
```
and r0,mail,#0xfffffff0
.unreq mail
pop {pc}
```

This code moves the answer (the top 28 bits of mail) to **r0**.

4 My Dearest Graphics Processor

Through our new postman, we now have the ability to send a message to the graphics card. What should we send though? This was certainly a difficult question for me to find the answer to, as it isn't in any online manual that I have found. Nevertheless, by looking at the GNU/Linux for the Raspberry Pi, we are able to work out what we needed to send.

> Since the RAM is shared between the graphics processor and the processor on the Pi, we can just send where to find our message. This is called DMA, many complicated devices use this to speed up access times.

The message is very simple. We describe the framebuffer we would like, and the graphics card either agrees to our request, in which case it sends us back a 0, and fills in a small questionnaire we make, or it sends back a non-zero number, in which case we know it is unhappy. Unfortunately, I have no idea what any of the other numbers it can send back are, nor what they mean, but only when it sends a zero it is happy. Fortunately it always seems to send a zero for sensible inputs, so we don't need to worry too much.

For simplicity we shall design our request in

advance, and store it in the .data section. In a file called 'framebuffer.s' place the following code:

```
.section .data
.align 4
.globl FrameBufferInfo
FrameBufferInfo:
.int 1024 /* #0 Physical Width */
.int 768 /* #4 Physical Height */
.int 1024 /* #8 Virtual Width */
.int 768 /* #12 Virtual Height */
.int 0 /* #16 GPU - Pitch */
.int 16 /* #20 Bit Depth */
.int 0 /* #24 X */
.int 0 /* #28 Y */
.int 0 /* #32 GPU - Pointer */
.int 0 /* #36 GPU - Size */
```

This is the format of our messages to the graphics processor. The first two words describe the physical width and height. The second pair is the virtual width and height. The framebuffer's width and height are the virtual width and height, and the GPU scales the framebuffer as need to fit the physical screen. The next word is one of the ones the GPU will fill in if it grants our request. It will be the number of bytes on each row of the frame buffer, in this case $2 \times 1024 = 2048$. The next word is how many bits to allocate to each pixel. Using a value of 16 means that the graphics processor uses High Colour mode described above. A value of 24 would use True Colour, and 32 would use RGBA32. The next two words are x and y offsets, which mean the number of pixels to skip in the top left corner of the screen when copying the framebuffer to the screen.

Finally, the last two words are filled in by the graphics processor, the first of which is the actual pointer to the frame buffer, and the second is the size of the frame buffer in bytes.

> When working with devices using DMA, alignment constraints become very important. The GPU expects the message to be 16 byte aligned.

I was very careful to include a **.align 4** here. As discussed before, this ensures the lowest 4 bits of the address of the next line are 0. Thus, we know for sure that FrameBufferInfo will be placed at an address we can send to the graphics processor, as our mailbox only sends values with the low 4 bits all 0.

So, now that we have our message, we can write code to send it. The communication will go as follows:

1. Write the address of FrameBufferInfo + 0x40000000 to mailbox 1.
2. Read the result from mailbox 1. If it is not zero, we didn't ask for a proper frame buffer.
3. Copy our images to the pointer, and they will appear on screen!

I've said something that I've not mentioned before in step 1. We have to add 0x40000000 to the address of FrameBufferInfo before sending it. This is actually a special signal to the GPU of how it should write to the structure. If we just send the address, the GPU will write its response, but will not make sure we can see it by flushing its cache. The cache is a piece of memory where a

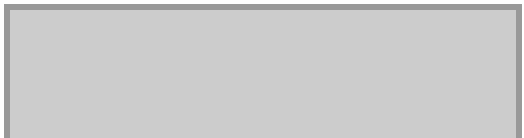
processor stores values its working on before sending them to the RAM. By adding 0x40000000, we tell the GPU not to use its cache for these writes, which ensures we will be able to see the change.

Since there is quite a lot going on there, it would be best to implement this as a function, rather than just putting the code into main.s. We shall write a function InitialiseFrameBuffer which does all this negotiation and returns the pointer to the frame buffer info data above, once it has a pointer in it. For ease, we should also make it so that the width, height and bit depth of the frame buffer are inputs to this method, so that it is easy to change in main.s without having to get into the details of the negotiation.

Once again, let's write down in detail the steps we will have to take. If you're feeling confident, try writing the function straight away.

1. Validate our inputs.
2. Write the inputs into the frame buffer.
3. Send the address of the frame buffer + 0x40000000 to the mailbox.
4. Receive the reply from the mailbox.
5. If the reply is not 0, the method has failed. We should return 0 to indicate failure.
6. Return a pointer to the frame buffer info.

Now we're getting into much bigger methods than before. Below is one implementation of the above.



1.
```
.section .text
.globl InitialiseFrameBuffer
InitialiseFrameBuffer:
width .req r0
height .req r1
bitDepth .req r2
cmp width,#4096
cmpls height,#4096
cmpls bitDepth,#32
result .req r0
movhi result,#0
movhi pc,lr
```

This code checks that the width and height are less than or equal to 4096, and that the bit depth is less than or equal to 32. This is once again using a trick with conditional execution. Convince yourself that this works.

2.
```
fbInfoAddr .req r3
push {lr}
ldr fbInfoAddr, =FrameBufferInfo
str width,[fbInfoAddr,#0]
str height,[fbInfoAddr,#4]
str width,[fbInfoAddr,#8]
str height,[fbInfoAddr,#12]
str bitDepth,[fbInfoAddr,#20]
.unreq width
.unreq height
.unreq bitDepth
```

This code simply writes into our frame buffer structure defined above. I also take the opportunity to push the link register onto the stack.

3.
```
mov r0,fbInfoAddr
```

```
add r0,#0x40000000
mov r1,#1
bl MailboxWrite
```

The inputs to the MailboxWrite method are the value to write in **r0**, and the channel to write to in **r1**.

4.
```
mov r0,#1
bl MailboxRead
```

The inputs to the MailboxRead method is the channel to write to in **r0**, and the output is the value read.

5.
```
teq result,#0
movne result,#0
popne {pc}
```

This code checks if the result of the MailboxRead method is 0, and returns 0 if not.

6.
```
mov result,fbInfoAddr
pop {pc}
.unreq result
.unreq fbInfoAddr
```

This code finishes off and returns the frame buffer info address.

5 A Pixel Within a Row Within a Frame

So, we've now created our methods to communicate with the graphics processor. It

should now be capable of giving us the pointer to a frame buffer we can draw graphics to. Let's draw something now.

In this first example, we'll just draw consecutive colours to the screen. It won't look pretty, but at least it will be working. How we will do this is by setting each pixel in the framebuffer to a consecutive number, and continually doing so.

Copy the following code to 'main.s' after **mov sp,#0x8000**

```
mov r0,#1024
mov r1,#768
mov r2,#16
bl InitialiseFrameBuffer
```

This code simply uses our InitialiseFrameBuffer method to create a frame buffer with width 1024, height 768, and bit depth 16. You can try different values in here if you wish, as long as you are consistent throughout the code. Since it's possible that this method can return 0 if the graphics processor did not give us a frame buffer, we had better check for this, and turn the OK LED on if it happens.

```
teq r0,#0
bne noError$

mov r0,#16
mov r1,#1
bl SetGpioFunction
mov r0,#16
mov r1,#0
bl SetGpio
```

```
error$:
b error$

noError$:
fbInfoAddr .req r4
mov fbInfoAddr,r0
```

Now that we have the frame buffer info address, we need to get the frame buffer pointer from it, and start drawing to the screen. We will do this using two loops, one going down the rows, and one going along the columns. On the Raspberry Pi, indeed in most applications, pictures are stored left to right then top to bottom, so we have to do the loops in the order I have said.

```
render$:
	fbAddr .req r3
	ldr fbAddr,[fbInfoAddr,#32]

	colour .req r0
	y .req r1
	mov y,#768
	drawRow$:
		x .req r2
		mov x,#1024
		drawPixel$:
			strh colour,[fbAddr]
			add fbAddr,#2
			sub x,#1
			teq x,#0
			bne drawPixel$

		sub y,#1
		add colour,#1
		teq y,#0
		bne drawRow$
```

```
        b render$

.unreq fbAddr
.unreq fbInfoAddr
```

strh reg, [dest] stores the low half word number in **reg** at the address given by **dest**.

This is quite a large chunk of code, and has a loop within a loop within a loop. To help get your head around the looping, I've indented the code which is looped, depending on which loop it is in. This is quite common in most high level programming languages, and the assembler simply ignores the tabs. We see here that I load in the frame buffer address from the frame buffer information structure, and then loop over every row, then every pixel on the row. At each pixel, I use an **strh** (store half word) command to store the current colour, then increment the address we're writing to. After drawing each row, we increment the colour that we are drawing. After drawing the full screen, we branch back to the beginning.

6 Seeing the Light

Now you're ready to test this code on the Raspberry Pi. You should see a changing gradient pattern. Be careful: until the first message is sent to the mailbox, the Raspberry Pi displays a still gradient pattern between the four corners. If it doesn't work, please see our troubleshooting page.

If it does work, congratulations! You can now

control the screen! Feel free to alter this code to draw whatever pattern you like. You can do some very nice gradient patterns, and can compute the value of each pixel directly, since **y** contains a y-coordinate for the pixel, and **x** contains an x-coordinate. In the next lesson, Lesson 7: Screen 02, we will look at one of the most common drawing tasks, lines.

Spot a mistake? You can help improve this tutorial on GitHub.

Lesson 7 Screen02

The Screen02 lesson builds on Screen01, by teaching how to draw lines and also a small feature on generating pseudo random numbers. It is assumed you have the code for the Lesson 6: Screen01 operating system as a basis.

1 Dots

Now that we've got the screen working, it is only natural to start waiting to create sensible images. It would be very nice indeed if we were able to actually draw something. One of the most basic components in all drawings is a line. If we were able to draw a line between any two points on the screen, we could start creating more complicated drawings just using combinations of these lines.

> To allow complex drawing, some systems use a colouring function rather than just one colour to draw things. Each pixel calls the colouring function to determine what colour to draw there.

We will attempt to implement this in assembly code, but first we could really use some other functions to help. We need a function I will call SetPixel that changes the colour of a particular pixel, supplied as inputs in r0 and r1. It will be helpful for future if we write code that could draw to any memory, not just the screen, so first of

all, we need some system to control where we are actually going to draw to. I think that the best way to do this would be to have a piece of memory which stores where we are going to draw to. What we should end up with is a stored address which normally points to the frame buffer structure from last time. We will use this at all times in our drawing method. That way, if we want to draw to a different image in another part of our operating system, we could make this value the address of a different structure, and use the exact same code. For simplicity we will use another piece of data to control the colour of our drawings.

Copy the following code to a new file called 'drawing.s'.

```
.section .data
.align 1
foreColour:
.hword 0xFFFF

.align 2
graphicsAddress:
.int 0

.section .text
.globl SetForeColour
SetForeColour:
cmp r0,#0x10000
movhs pc,lr
ldr r1,=foreColour
strh r0,[r1]
mov pc,lr

.globl SetGraphicsAddress
SetGraphicsAddress:
ldr r1,=graphicsAddress
```

```
str r0,[r1]
mov pc,lr
```

This is just the pair of functions that I described above, along with their data. We will use them in 'main.s' before drawing anything to control where and what we are drawing.

> Building generic methods like SetPixel which we can build other methods on top of is a useful idea. We have to make sure the method is fast though, since we will use it a lot.

Our next task is to implement a SetPixel method. This needs to take two parameters, the x and y co-ordinate of a pixel, and it should use the graphicsAddress and foreColour we have just defined to control exactly what and where it is drawing. If you think you can implement this immediately, do, if not I shall outline the steps to be taken, and then give an example implementation.

1. Load in the graphicsAddress.
2. Check that the x and y co-ordinates of the pixel are less than the width and height.
3. Compute the address of the pixel to write. (hint: frameBufferAddress + (x + y * width) * pixel size)
4. Load in the foreColour.
5. Store it at the address.

An implementation of the above follows.

1.
```
.globl DrawPixel
DrawPixel:
px .req r0
py .req r1
```

```
addr .req r2
ldr addr,=graphicsAddress
ldr addr,[addr]
```

2.
```
height .req r3
ldr height,[addr,#4]
sub height,#1
cmp py,height
movhi pc,lr
.unreq height

width .req r3
ldr width,[addr,#0]
sub width,#1
cmp px,width
movhi pc,lr
```

Remember that the width and height are stored at offsets of 0 and 4 into the frame buffer description respectively. You can refer back to 'frameBuffer.s' if necessary.

3.
```
ldr addr,[addr,#32]
add width,#1
mla px,py,width,px
.unreq width
.unreq py
add addr, px,lsl #1
.unreq px
```

mla dst,reg1,reg2,reg3 multiplies the values from **reg1** and **reg2**, adds the value from **reg3** and places the least significant 32 bits of the result in **dst**.

Admittedly, this code is specific to high colour frame buffers, as I use a bit shift directly to compute this address. You

may wish to code a version of this function without the specific requirement to use high colour frame buffers, remembering to update the SetForeColour code. It may be significantly more complicated to implement.

4.
```
fore .req r3
ldr fore, =foreColour
ldrh fore,[fore]
```

   As above, this is high colour specific.

5.
```
strh fore,[addr]
.unreq fore
.unreq addr
mov pc,lr
```

   As above, this is high colour specific.

2 Lines

The trouble is, line drawing isn't quite as simple as you may expect. By now you must realise that when making operating system, we have to do almost everything ourselves, and line drawing is no exception. I suggest for a few minutes you have a think about how you would draw a line between any two points.

> When programming normally, we tend to be lazy with things like division. Operating Systems must be incredibly efficient, and so we must focus on doing things as best as possible.

I expect the central idea of most strategies

will involve computing the gradient of the line, and stepping along it. This sounds perfectly reasonable, but is actually a terrible idea. The problem with it is it involves division, which is something that we know can't easily be done in assembly, and also keeping track of decimal numbers, which is again difficult. There is, in fact, an algorithm called Bresenham's Algorithm, which is perfect for assembly code because it only involves addition, subtraction and bit shifts.

Algorithm derivation

Let's start off by defining a reasonably straightforward line drawing algorithm as follows:

```
/* We wish to draw a line from
(x0,y0) to (x1,y1), using only a
function setPixel(x,y) which draws a
dot in the pixel given by (x,y). */
if x1 > x0 then
      set deltax to x1 - x0
      set stepx to +1
otherwise
      set deltax to x0 - x1
      set stepx to -1
end if

if y1 > y0 then
      set deltay to y1 - y0
      set stepy to +1
otherwise
      set deltay to y0 - y1
      set stepy to -1
end if
```

```
if deltax > deltay then
    set error to 0
    until x0 = x1 + stepx
        setPixel(x0, y0)
        set error to error + deltax
        ÷ deltay
        if error ≥ 0.5 then
            set y0 to y0 + stepy
            set error to error - 1
        end if
        set x0 to x0 + stepx
    repeat
otherwise
    /* As above but swap x and y
    */
end if
```

This algorithm is a representation of the sort of thing you may have imagined. The variable error keeps track of how far away from the actual line we are. Every step we take along the x axis increases this error, and every time we move down the y axis, the error decreases by 1 unit again. The error is measured as a distance along the y axis.

While this algorithm works, it suffers a major problem in that we clearly have to use decimal numbers to store error, and also we have to do a division. An immediate optimisation would therefore be to change the units of error. There is no need to store it in any particular units, as long as we scale every use of it by the same amount. Therefore, we could rewrite the algorithm simply by multiplying all equations involving error by deltay,

and simplifying the result. Just showing the main loop:

```
set error to 0 × deltay
until x0 = x1 + stepx
    setPixel(x0, y0)
    set error to error + deltax ÷
    deltay × deltay
    if error ≥ 0.5 × deltay then
        set y0 to y0 + stepy
        set error to error - 1 ×
        deltay
    end if
    set x0 to x0 + stepx
repeat
```

Which simplifies to:

```
set error to 0
until x0 = x1 + stepx
    setPixel(x0, y0)
    set error to error + deltax
    if error × 2 ≥ deltay then
        set y0 to y0 + stepy
        set error to error - deltay
    end if
    set x0 to x0 + stepx
repeat
```

Suddenly we have a much better algorithm. We see now that we've eliminated the need for division altogether. Better still, the only multiplication is by 2, which we know is just a bit shift left by 1! This is now very close to Bresenham's Algorithm, but one further optimisation can be made. At the moment, we have an if

statement which leads to two very similar blocks of code, one for lines with larger x differences, and one for lines with larger y differences. It is worth checking if the code could be converted to a single statement for both types of line.

The difficulty arises somewhat in that in the first case, error is to do with y, and in the second case error is to do with x. The solution is to track the error in both variables simultaneously, using negative error to represent an error in x, and positive error in y.

```
set error to deltax - deltay
until x0 = x1 + stepx or y0 = y1
+ stepy
    setPixel(x0, y0)
    if error × 2 > -deltay then
        set x0 to x0 + stepx
        set error to error - deltay
    end if
    if error × 2 < deltax then
        set y0 to y0 + stepy
        set error to error + deltax
    end if
repeat
```

It may take some time to convince yourself that this actually works. At each step, we consider if it would be correct to move in x or y. We do this by checking if the magnitude of the error would be lower if we moved in the x or y co-ordinates, and then moving if so.

Bresenham's Line Algorithm was developed in 1962 by Jack Elton Bresenham, 24 at the time, whilst studying for a PhD.

Bresenham's Algorithm for drawing a line can be described by the following pseudo code. Pseudo code is just text which looks like computer instructions, but is actually intended for programmers to understand algorithms, rather than being machine readable.

```
/* We wish to draw a line from (x0,y0) to
(x1,y1), using only a function setPixel(x,y)
which draws a dot in the pixel given by
(x,y). */
if x1 > x0 then
    set deltax to x1 - x0
    set stepx to +1
otherwise
    set deltax to x0 - x1
    set stepx to -1
end if

set error to deltax - deltay
until x0 = x1 + stepx or y0 = y1 +
stepy
    setPixel(x0, y0)
    if error × 2 ≥ -deltay then
        set x0 to x0 + stepx
        set error to error - deltay
    end if
    if error × 2 ≤ deltax then
        set y0 to y0 + stepy
        set error to error + deltax
    end if
repeat
```

Rather than numbered lists as I have used so

far, this representation of an algorithm is far more common. See if you can implement this yourself. For reference, I have provided my implementation below.

```
.globl DrawLine
DrawLine:
push {r4,r5,r6,r7,r8,r9,r10,r11,r12,lr}
x0 .req r9
x1 .req r10
y0 .req r11
y1 .req r12

mov x0,r0
mov x1,r2
mov y0,r1
mov y1,r3

dx .req r4
dyn .req r5 /* Note that we only ever use -deltay, so I store its negative for speed. (hence dyn) */
sx .req r6
sy .req r7
err .req r8

cmp x0,x1
subgt dx,x0,x1
movgt sx,#-1
suble dx,x1,x0
movle sx,#1

cmp y0,y1
subgt dyn,y1,y0
movgt sy,#-1
suble dyn,y0,y1
movle sy,#1

add err,dx,dyn
add x1,sx
```

```
add y1,sy

pixelLoop$:
	teq x0,x1
	teqne y0,y1
	popeq
	{r4,r5,r6,r7,r8,r9,r10,r11,r12,pc}

	mov r0,x0
	mov r1,y0
	bl DrawPixel

	cmp dyn, err,lsl #1
	addle err,dyn
	addle x0,sx

	cmp dx, err,lsl #1
	addge err,dx
	addge y0,sy

	b pixelLoop$

.unreq x0
.unreq x1
.unreq y0
.unreq y1
.unreq dx
.unreq dyn
.unreq sx
.unreq sy
.unreq err
```

3 Randomness

So, now we can draw lines. Although we could use this to draw pictures and whatnot (feel free to do so!), I thought I would take the opportunity to introduce the idea of computer randomness. What we will do is

select a pair of random co-ordinates, and then draw a line from the last pair to that point in steadily incrementing colours. I do this purely because it looks quite pretty.

> Hardware random number generators are occasionally used in security, where the predictability sequence of random numbers may affect the security of some encryption.

So, now it comes down to it, how do we be random? Unfortunately for us there isn't some device which generates random numbers (such devices are very rare). So somehow using only the operations we've learned so far we need to invent 'random numbers'. It shouldn't take you long to realise this is impossible. The operations always have well defined results, executing the same sequence of instructions with the same registers yields the same answer. What we instead do is deduce a sequence that is pseudo random. This means numbers that, to the outside observer, look random, but in fact were completely determined. So, we need a formula to generate random numbers. One might be tempted to just spam mathematical operators out for example: $4x^2! / 64$, but in actuality this generally produces low quality random numbers. In this case for example, if x were 0, the answer would be 0. Stupid though it sounds, we need a very careful choice of equation to produce high quality random numbers.

> This sort of discussion often begs the question what do we mean by a random number? We generally mean statistical randomness: A sequence of numbers that has no obvious patterns or properties that could be used to generalise it.

The method I'm going to teach you is called the quadratic congruence generator. This is a good choice because it can be implemented in 5 instructions, and yet generates a seemingly random order of the numbers from 0 to 2^{32}-1.

The reason why the generator can create such long sequence with so few instructions is unfortunately a little beyond the scope of this course, but I encourage the interested to research it. It all centralises on the following quadratic formula, where x_n is the nth random number generated.

$$x_{n+1} = ax_n^2 + bx_n + c \bmod 2^{32}$$

Subject to the following constraints:

1. a is even
2. $b = a + 1 \bmod 4$
3. c is odd

If you've not seen mod before, it means the remainder of a division by the number after it. For example $b = a + 1 \bmod 4$ means that b is the remainder of dividing $a + 1$ by 4, so if a were 12 say, b would be 1 as $a + 1$ is 13, and 13 divided by 4 is 3 remainder 1.

Copy the following code into a file called 'random.s'.

```
.globl Random
Random:
xnm .req r0
a .req r1

mov a,#0xef00
mul a,xnm
mul a,xnm
```

```
add a,xnm
.unreq xnm
add r0,a,#73

.unreq a
mov pc,lr
```

This is an implementation of the random function, with an input of the last value generated in **r0**, and an output of the next number. In my case, I've used a = $EF00_{16}$, b = 1, c = 73. This choice was arbitrary but meets the requirements above. Feel free to use any numbers you wish instead, as long as they obey the rules.

4 Pi-casso

OK, now we have all the functions we're going to need, let's try it out. Alter main to do the following, after getting the frame buffer info address:

1. Call SetGraphicsAddress with r0 containing the frame buffer info address.
2. Set four registers to 0. One will be the last random number, one will be the colour, one will be the last x co-ordinate and one will be the last y co-ordinate.
3. Call random to generate the next x-coordinate, using the last random number as the input.
4. Call random again to generate the next y-coordinate, using the x-coordinate you generated as an input.
5. Update the last random number to contain the y-coordinate.

6. Call SetForeColour with the colour, then increment the colour. If it goes above $FFFF_{16}$, make sure it goes back to 0.
7. The x and y coordinates we have generated are between 0 and $FFFFFFFF_{16}$. Convert them to a number between 0 and 1023_{10} by using a logical shift right of 22.
8. Check the y coordinate is on the screen. Valid y coordinates are between 0 and 767_{10}. If not, go back to 3.
9. Draw a line from the last x and y coordinates to the current x and y coordinates.
10. Update the last x and y coordinates to contain the current ones.
11. Go back to 3.

As always, a solution for this can be found on the downloads page.

Once you've finished, test it on the Raspberry Pi. You should see a very fast sequence of random lines being drawn on the screen, in steadily incrementing colours. This should never stop. If it doesn't work, please see our troubleshooting page.

When you have it working, congratulations! We've now learned about meaningful graphics, and also about random numbers. I encourage you to play with line drawing, as it can be used to render almost anything you want. You may also want to explore more complicated shapes. Most can be made out of lines, but is this necessarily the best strategy? If you like the line program, try experimenting with the SetPixel function.

What happens if instead of just setting the value of the pixel, you increase it by a small amount? What other patterns can you make? In the next lesson, Lesson 8: Screen 03, we will look at the invaluable skill of drawing text.

Spot a mistake? You can help improve this tutorial on GitHub.

Lesson 8 Screen03

The Screen03 lesson builds on Screen02, by teaching how to draw text, and also a small feature on the command line arguments of the operating system. It is assumed you have the code for the Lesson 7: Screen02 operating system as a basis.

1 String Theory

So, our task for this operating system is to draw text. We have several problems to address, the most pressing of which is probably about storing text. Unbelievably text has been one of the biggest flaws on computers to date. What should have been a straightforward type of data has brought down operating systems, crippled otherwise wonderful encryption, and caused many problems for users of different alphabets. Nevertheless, it is an incredibly important type of data, as it is an excellent link between the computer and the user. Text can be sufficiently structured that the operating system understands it, as well as sufficiently readable that humans can use it.

> Variable data types such as text require much more complex handling.

So how exactly is text stored? Simply enough, we have some system by which we give each letter a unique number, and then store a sequence of such numbers. Sounds easy. The

problem is that the number of numbers is not fixed. Some pieces of text are longer than others. With storing ordinary numbers, we have some fixed limit, e.g. 32 bits, and then we can't go beyond that, we write methods that use numbers of that length, etc. In terms of text, or strings as we often call it, we want to write functions that work on variable length strings, otherwise we would need a lot of functions! This is not a problem for numbers normally, as there are only a few common number formats (byte, word, half, double).

> Buffer overrun attacks have plagued computers for years. Recently, the Wii, Xbox and Playstation 2 all suffered buffer overrun attacks, as well as large systems like Microsoft's Web and Database servers.

So, how do we determine how long the string is? I think the obvious answer is just to store how long the string is, and then to store the characters that make it up. This is called length prefixing, as the length comes before the string. Unfortunately, the pioneers of computer science did not agree. They felt it made more sense to have a special character called the null terminator (denoted \0) which represents when a string ends. This does indeed simplify many string algorithms, as you just keep working until the null terminator. Unfortunately this is the source of many security issues. What if a malicious user gives you a very long string? What if you didn't have enough space to store it. You might run a string copying function that copies until the null terminator, but because the string is so long, it overwrites your program. This may sound far fetched, but

nevertheless, such buffer overrun attacks are incredibly common. Length prefixing mitigates this problem as it is easy to deduce the size of the buffer required to store the string. As an operating system developer, I leave it to you to decide how best to store text.

The next thing we need to establish is how best to map characters to numbers. Fortunately, this is reasonably well standardised, so you have two major choices, Unicode and ASCII. Unicode maps almost every single useful symbol that can be written to a number, in exchange for having a lot more numbers, and a more complicated encoding system. ASCII uses one byte per character, and so only stores the Latin alphabet, numbers, a few symbols and a few special characters. Thus, ASCII is very easy to implement, compared to Unicode, in which not every character takes the same space, making string algorithms tricky. Normally operating systems use ASCII for strings which will not be displayed to end users (but perhaps to developers or experts), and Unicode for displaying messages to users, as Unicode can support things like Japanese characters, and so could be localised.

Fortunately for us, this decision is irrelevant at the moment, as the first 128 characters of both are exactly the same, and are encoded exactly the same.

```
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0(NI SO ST ET EC EN AC BE  B  H  L  V  F  C  S( SI
1( DI DC DC DC DC DC NA SY ET CA EI SU ES  F  G  R US
20     !  "  #  $  %  &  .  (  )  *  +  ,  -  .  /
30 0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
```

40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Table 1.1 ASCII/Unicode symbols 0-127

The table shows the first 128 symbols. The hexadecimal representation of the number for a symbol is the row value added to the column value, for example A is 41_{16}. What you may find surprising is the first two rows, and the very last value. These 33 special characters are not printed at all. In fact, these days, many are ignored. They exist because ASCII was originally intended as a system for transmitting data over computer networks, and so a lot more information than just the symbols had to be sent. The key special symbols that you should learn are NUL, the null terminator character I mentioned before, HT, horizontal tab is what we normally refer to as a tab and LF, the line feed character is used to make a new line. You may wish to research and use the other characters for special meanings in your operating system.

2 Characters

So, now that we know a bit about strings, we can start to think about how they're displayed. The fundamental thing we need to do in order to be able to display a string is to be able to display a character. Our first task will be making a DrawCharacter function which takes in a character to draw and a location, and then draws the character.

The true type font format used in many Operating Systems is so powerful, it has its own assembly language built in to ensure

letters look correct at any resolution.

Naturally, this leads to a discussion about fonts. We already know there are many ways to display any given letter in accordance with font choice. So how does a font work? In the very early days of computer science, a font was just a series of little pictures of all the letters, called a bitmap font, and all the draw character method would do is copy one of the pictures to the screen. The trouble with this is when people want to resize the text. Sometimes we need big letters, and sometimes small. Although we could keep drawing new pictures for every font at every size with every character, this would get tedious. Thus, vector fonts were invented. in vector fonts, rather than containing an image of the font, the font file contains a description of how to draw it, e.g. an 'o' could be circle with radius half that of the maximum letter height. Modern operating systems use such fonts almost exclusively, as they are perfect at any resolution.

Unfortunately, much though I would love to include an implementation of one of the vector font formats, it would take up the remainder of this website. Thus, we will implement a bitmap font, though if you wish to make a decent graphical operating system, a vector font would be useful.

On the downloads page, I have included several '.bin' files in the font section. These are just raw binary data files for a few fonts. For this tutorial, pick your favourite from the monospace, monochrome, 8x16 section. Download it and store it in the 'source' directory as 'font.bin'. These files are just monochrome images of each of the letters in turn, with each letter being exactly 8 by 16 pixels. Thus, each takes 16 bytes, the first byte being the top row, the second the next, etc.

The diagram shows the 'A' character in the monospace, monochrome, 8x16 font Bitstream Vera Sans Mono. In the file, we would find this starting at the $41_{16} \times 10_{16} = 410_{16}$th byte as the following sequence in hexadecimal:

00, 00, 00, 10, 28, 28, 28, 44, 44, 7C, C6, 82, 00, 00, 00, 00

We're going to use a monospace font here, because in a monospace font every character is the same size. Unfortunately, yet another complication with most fonts is that the character's widths vary, leading to more complex display code. I've included a few other fonts on the downloads page, as well as an explanation of the format I've stored them all in.

So let's get down to business. Copy the following to 'drawing.s' after the **.int 0** of graphicsAddress.

```
.align 4
font:
.incbin "font.bin"
```

.incbin "file" inserts the binary data from the file **file**.

This code copies the font data from the file to the address labelled font. We've used an **.align 4** here to ensure each character starts on a multiple of 16 bytes, which can be used for a speed trick later.

Now we want to write the draw character method. I'll give the pseudo code for this, so you can try to implement it yourself if you want to. Conventionally > > means logical shift right.

```
function drawCharacter(r0 is character, r1
is x, r2 is y)
    if character > 127 then exit
    set charAddress to font + character
    × 16
    for row = 0 to 15
        set bits to readByte(charAddress
        + row)
        for bit = 0 to 7
            if test(bits > > bit, 0x1)
            then setPixel(x + bit, y +
            row)
        next
    next
    return r0 = 8, r1 = 16
end function
```

If implemented directly, this is deliberately not very efficient. With things like drawing characters, efficiency is a top priority, as we will do it a lot. Let's explore some improvements that bring this closer to optimal assembly code. Firstly, we have a $\times$ 16, which by now you should spot is the same as a logical shift left by 4 places. Next we have a variable row, which is only ever added to charAddress and to y. Thus, we can eliminate it by increasing these variables instead. The only issue now is how to tell when we've finished. This is where the **.align 4** comes in handy. We know that charAddress will start with the low nibble containing 0. This means we can see how far into the character data we are by checking that low nibble.

Though we can eliminate the need for bits, we must introduce a new variable to do so, so it is best left in. The only other improvement that can be made is to remove the nested bits >> bit.

```
function drawCharacter(r0 is character, r1
is x, r2 is y)
    if character > 127 then exit
    set charAddress to font + character
    << 4
    loop
        set bits to readByte(charAddress)
        set bit to 8
        loop
            set bits to bits << 1
            set bit to bit - 1
            if test(bits, 0x100)
            then setPixel(x + bit, y)
        until bit = 0
        set y to y + 1
```

```
            set chadAddress to chadAddress
            + 1
        until charAddress AND 0b1111 = 0
        return r0 = 8, r1 = 16
end function
```

Now we've got code that is much closer to assembly code, and is near optimal. Below is the assembly code version of the above.

```
.globl DrawCharacter
DrawCharacter:
cmp r0,#127
movhi r0,#0
movhi r1,#0
movhi pc,lr

push {r4,r5,r6,r7,r8,lr}
x .req r4
y .req r5
charAddr .req r6
mov x,r1
mov y,r2
ldr charAddr,=font
add charAddr, r0,lsl #4

lineLoop$:
        bits .req r7
        bit .req r8
        ldrb bits,[charAddr]
        mov bit,#8

        charPixelLoop$:
                subs bit,#1
                blt charPixelLoopEnd$
                lsl bits,#1
                tst bits,#0x100
                beq charPixelLoop$
```

```
            add r0,x,bit
            mov r1,y
            bl DrawPixel

            teq bit,#0
            bne charPixelLoop$
        charPixelLoopEnd$:
        .unreq bit
        .unreq bits
        add y,#1
        add charAddr,#1
        tst charAddr,#0b1111
        bne lineLoop$
.unreq x
.unreq y
.unreq charAddr

width .req r0
height .req r1
mov width,#8
mov height,#16

pop {r4,r5,r6,r7,r8,pc}
.unreq width
.unreq height
```

3 Strings

Now that we can draw characters, we can draw text. We need to make a method that, for a given string, draws each character in turn, at incrementing positions. To be nice, we shall also implement new lines and tabs. It's decision time as far as null terminators are concerned, and if you want to make your operating system use them, feel free by changing the code below. To avoid the issue, I will have the length of the string passed as an argument to the DrawString function,

along with the address of the string, and the x and y coordinates.

```
function drawString(r0 is string, r1 is
length, r2 is x, r3 is y)
      set x0 to x
      for pos = 0 to length - 1
            set char to loadByte(string +
            pos)
            set (cwidth, cheight) to
            DrawCharacter(char, x, y)
            if char = '\n' then /* \n is short
            hand for the character LF */
                  set x to x0
                  set y to y + cheight
            otherwise if char = '\t' then /* \t
            is short hand for the character
            HT */
                  set x1 to x
                  until x1 > x0
                        set x1 to x1 + 5 ×
                        cwidth
                  loop
                  set x to x1
            otherwise
                  set x to x + cwidth
            end if
      next
end function
```

Once again, this function isn't that close to assembly code. Feel free to try to implement it either directly or by simplifying it. I will give the simplification and then the assembly code below.

Clearly the person who wrote this function wasn't being very efficient (me in case you were wondering). Once again we have a pos

variable that just increments and is added to something else, which is completely unnecessary. We can remove it, and instead simultaneously decrement length until it is 0, saving the need for one register. The rest of the function is probably fine, except for that annoying multiplication by five. A key thing to do here would be to move the multiplication outside the loop; multiplication is slow even with bit shifts, and since we're always adding the same constant multiplied by 5, there is no need to recompute this. It can in fact be implemented in one operation using the argument shifting in assembly code, so I shall rephrase it like that.

```
function drawString(r0 is string, r1 is
length, r2 is x, r3 is y)
    set x0 to x
    until length = 0
        set length to length - 1
        set char to loadByte(string)
        set (cwidth, cheight) to
        DrawCharacter(char, x, y)
        if char = '\n' then
            set x to x0
            set y to y + cheight
        otherwise if char = '\t' then
            set x1 to x
            set cwidth to cwidth +
            cwidth << 2
            until x1 > x0
                set x1 to x1 + cwidth
            loop
            set x to x1
        otherwise
            set x to x + cwidth
        end if
        set string to string + 1
```

```
    loop
end function
```

In assembly code:

```
.globl DrawString
DrawString:
x .req r4
y .req r5
x0 .req r6
string .req r7
length .req r8
char .req r9
push {r4,r5,r6,r7,r8,r9,lr}

mov string,r0
mov x,r2
mov x0,x
mov y,r3
mov length,r1

stringLoop$:
    subs length,#1
    blt stringLoopEnd$

    ldrb char,[string]
    add string,#1

    mov r0,char
    mov r1,x
    mov r2,y
    bl DrawCharacter
    cwidth .req r0
    cheight .req r1

    teq char,#'\n'
    moveq x,x0
    addeq y,cheight
    beq stringLoop$
```

```
        teq char,#'\t'
        addne x,cwidth
        bne stringLoop$

        add cwidth, cwidth,lsl #2
        x1 .req r1
        mov x1,x0

        stringLoopTab$:
                add x1,cwidth
                cmp x,x1
                bge stringLoopTab$
        mov x,x1
        .unreq x1
        b stringLoop$
stringLoopEnd$:
.unreq cwidth
.unreq cheight

pop {r4,r5,r6,r7,r8,r9,pc}
.unreq x
.unreq y
.unreq x0
.unreq string
.unreq length
```

subs reg,#val subtracts **val** from the register **reg** and compares the result with 0.

This code makes clever use of a new operation, **subs** which subtracts one number from another, stores the result and then compares it with 0. In truth, all comparisons are implemented as a subtraction and then comparison with 0, but the result is normally discarded. This means that this operation is as fast as **cmp**.

4 Your Wish is My Command Line

Now that we can print strings, the challenge is to find an interesting one to draw. Normally in tutorials such as this, people just draw "Hello World!", but after all we've done so far, I feel that is a little patronising (feel free to do so if it helps). Instead we're going to draw our command line.

A convention has been made for computers running ARM. When they boot, it is important they are given certain information about what they have available to them. Most all processors have some way of ascertaining this information, and on ARM this is by data left at the address 100_{16}. The format of the data is as follows:

1. The data is broken down into a series of 'tags'.
2. There are nine types of tag: 'core', 'mem', 'videotext', 'ramdisk', 'initrd2', 'serial' 'revision', 'videolfb', 'cmdline'.
3. Each can only appear once, but all but the 'core' tag don't have to appear.
4. The tags are placed from 0x100 in order one after the other.
5. The end of the list of tags always contains 2 words which are 0.
6. Every tag's size in bytes is a multiple of 4.
7. Each tag starts with the size of the tag in words in the tag, including this number.
8. This is followed by a half word containing the tag's number. These are numbered from 1 in the order above

('core' is 1, 'cmdline' is 9).
9. This is followed by a half word containing 5441_{16}.
10. After this comes the data of the tag, which varies depending on the tag. The size of the data in words + 2 is always the same as the length mentioned above.
11. A 'core' tag is either 2 or 5 words in length. If it is 2, there is no data, if it is 5, it has 3 words.
12. A 'mem' tag is always 4 words in length. The data is the first address in a block of memory, and the length of that block.
13. A 'cmdline' tag contains a null terminated string which is the parameters of the kernel.

> Almost all Operating Systems support the notion of programs having a 'command line'. The idea is to provide a common mechanism for choosing the desired behaviour of the program.

On the current version of the Raspberry Pi, only the 'core', 'mem' and 'cmdline' tags are present. You may find these useful later, and a more complete reference for these is on our Raspberry Pi reference page. The one we're interested in at the moment is the 'cmdline' tag, because it contains a string. We're going to write some code to search for the command line tag, and, if found, to print it out with each item on a new line. The command line is just a list of things that either the graphics processor or the user thought it might be nice for the Operating System to know. On the Raspberry Pi, this includes the MAC Address, serial number and screen resolution. The string itself is just a list

of expressions such as 'key.subkey=value' separated by spaces.

Let's start by finding the 'cmdline' tag. In a new file called 'tags.s' copy the following code.

```
.section .data
tag_core: .int 0
tag_mem: .int 0
tag_videotext: .int 0
tag_ramdisk: .int 0
tag_initrd2: .int 0
tag_serial: .int 0
tag_revision: .int 0
tag_videolfb: .int 0
tag_cmdline: .int 0
```

Looking through the list of tags will be a slow operation, as it involves a lot of memory access. Therefore, we only want to have to do it once. This code creates some data which will store the memory address of the first tag of each of the types. Then, to find a tag the following pseudo code will suffice.

```
function FindTag(r0 is tag)
    if tag > 9 or tag = 0 then return 0
    set tagAddr to loadWord(tag_core +
    (tag - 1) × 4)
    if not tagAddr = 0 then return
    tagAddr
    if readWord(tag_core) = 0 then
    return 0
    set tagAddr to 0x100
    loop forever
        set tagIndex to
        readHalfWord(tagAddr + 4)
        if tagIndex = 0 then return
```

```
            FindTag(tag)
            if readWord(tag_core
            +(tagIndex-1)×4) = 0
            then storeWord(tagAddr, tag_core
            +(tagIndex-1)×4)
            set tagAddr to tagAddr +
            loadWord(tagAddr) × 4
        end loop
end function
```

This code is already quite well optimised and close to assembly. It is optimistic in that the first thing it tries is loading the tag directly, as all but the first time this should be the case. If that fails, it checks if the core tag has an address. Since there must always be a core tag, the only reason that it would not have an address is if it doesn't exist. If it does have an address, the tag we were looking for didn't. If it doesn't we need to find the addresses of all the tags. It does this by reading the number of the tag. If it is zero, that must mean we are at the end of the list. This means we've now filled in all the tags in our directory. Therefore if we run our function again, it will now be able to produce an answer. If the tag number is not zero, we check to see if this tag type already has an address. If not, we store the address of this tag in our directory. We then add the length of this tag in bytes to the tag address to find the next tag.

Have a go at implementing this code in assembly. You will need to simplify it. If you get stuck, my answer is below. Don't forget the **.section .text**!

```
.section .text
.globl FindTag
```

```
FindTag:
tag .req r0
tagList .req r1
tagAddr .req r2

sub tag,#1
cmp tag,#8
movhi tag,#0
movhi pc,lr

ldr tagList, =tag_core
tagReturn$:
add tagAddr,tagList, tag,lsl #2
ldr tagAddr,[tagAddr]

teq tagAddr,#0
movne r0,tagAddr
movne pc,lr

ldr tagAddr,[tagList]
teq tagAddr,#0
movne r0,#0
movne pc,lr

mov tagAddr,#0x100
push {r4}
tagIndex .req r3
oldAddr .req r4
tagLoop$:
ldrh tagIndex,[tagAddr,#4]
subs tagIndex,#1
poplt {r4}
blt tagReturn$

add tagIndex,tagList, tagIndex,lsl #2
ldr oldAddr,[tagIndex]
teq oldAddr,#0
.unreq oldAddr
streq tagAddr,[tagIndex]
```

```
ldr tagIndex,[tagAddr]
add tagAddr, tagIndex,lsl #2
b tagLoop$

.unreq tag
.unreq tagList
.unreq tagAddr
.unreq tagIndex
```

5 Hello World

Now that we have everything we need, we can draw our first string. In 'main.s' delete everything after **bl SetGraphicsAddress**, and replace it with the following:

```
mov r0,#9
bl FindTag
ldr r1,[r0]
lsl r1,#2
sub r1,#8
add r0,#8
mov r2,#0
mov r3,#0
bl DrawString
loop$:
b loop$
```

This code simply uses our FindTag method to find the 9th tag (cmdline) and then calculates its length and passes the command and the length to the DrawString method, and tells it to draw the string at 0,0. Now test this on the Raspberry Pi. You should see a line of text on the screen. If not please see our troubleshooting page.

Once it works, congratulations you've now

got the ability to draw text. But there is still room for improvement. What if we wanted to write out a number, or a section of the memory or manipulate our command line? In Lesson 9: Screen04, we will look at manipulating text and displaying useful numbers and information.

Spot a mistake? You can help improve this tutorial on GitHub.

Lesson 9 Screen04

The Screen04 lesson builds on Screen03, by teaching how to manipulate text. It is assumed you have the code for the Lesson 8: Screen03 operating system as a basis.

1 String Manipulation

> Variadic functions look much less intuitive in assembly code. Nevertheless, they are useful and powerful concepts.

Being able to draw text is lovely, but unfortunately at the moment you can only draw strings which are already prepared. This is fine for displaying something like the command line, but ideally we would like to be able to display and text we so desire. As per usual, if we put the effort in and make an excellent function that does all the string manipulation we could ever want, we get much easier code later on in return. Once such complicated function in C programming is sprintf. This function generates a string based on a description given as another string and additional arguments. What is interesting about this function is that it is variadic. This means that it takes a variable number of parameters. The number of parameters depends on the exact format string, and so cannot be determined in advance.

The full function has many options, and I list a few here. I've highlighted the ones which

we will implement in this tutorial, though you can try to implement more.

The function works by reading the format string, and then interpreting it using the table below. Once an argument is used, it is not considered again. The return value of the function is the number of characters written. If the method fails, a negative number is returned.

Sequence	Meaning
Any character except %	Copies the character to the output.
%%	Writes a % character to the output.
%c	Writes the next argument as a character.
%d *or* %i	Writes the next argument as a base 10 signed integer.
%e	Writes the next argument in scientific notation using e**N** to mean $\times 10^{N}$.
%E	Writes the next argument in scientific notation using E**N** to mean $\times 10^{N}$.
%f	Writes the next argument as a decimal IEEE 754 floating point number.
%g	Same as the shorter of %e and %f.
%G	Same as the shorter of %E and %f.
%o	Writes the next argument as a base 8

	unsigned integer.
%s	Writes the next argument as if it were a pointer to a null terminated string.
%u	Writes the next argument as a base 10 unsigned integer.
%x	Writes the next argument as a base 16 unsigned integer, with lowercase a,b,c,d,e and f.
%X	Writes the next argument as a base 16 unsigned integer, with uppercase A,B,C,D,E and F.
%p	Writes the next argument as a pointer address.
%n	Writes nothing. Copies instead the number of characters written so far to the location addressed by the next argument.

Table 1.1 sprintf formatting rules

Further to the above, many additional tweaks exist to the sequences, such as specifying minimum length, signs, etc. More information can be found at sprintf - C++ Reference.

Here are a few examples of calls to the method and their results to illustrate its use.

Format String	**Arguments**	**Result**
"%d"	13	"13"

"+%d degrees"	12	"+12 degrees"
"+%x degrees"	24	"+1c degrees"
"'%c' = 0%o"	65, 65	"'A' = 0101"
"%d * %d%% = %d"	200, 40, 80	"200 * 40% = 80"
"+%d degrees"	-5	"+-5 degrees"
"+%u degrees"	-5	"+4294967291 degrees"

Table 1.2 sprintf example calls

Hopefully you can already begin to see the usefulness of the function. It does take a fair amount of work to program, but our reward is a very general function we can use for all sorts of purposes.

2 Division

> Division is the slowest and most complicated of the basic mathematical operators. It is not implemented directly in ARM assembly code because it takes so long to deduce the answer, and so isn't a 'simple' operation.

While this function does look very powerful, it also looks very complicated. The easiest way to deal with its many cases is probably to write functions to deal with some common tasks it has. What would be useful would be a function to generate the string for a signed and an unsigned number in any base. So, how can we go about doing that? Try to devise an algorithm quickly before reading on.

The easiest way is probably the exact way I mentioned in Lesson 1: OK01, which is the division remainder method. The idea is the following:

1. Divide the current value by the base you're working in.
2. Store the remainder.
3. If the new value is not 0, go to 1.
4. Reverse the order of the remainders. This is the answer.

For example:

Value	New Value	Remainder
137	68	1
68	34	0
34	17	0
17	8	1
8	4	0
4	2	0
2	1	0
1	0	1

Table 2.1 Example base 2 conversion

So the answer is 10001001_2

The unfortunate part about this procedure is that it unavoidably uses division. Therefore, we must first contemplate division in binary.

For a refresher on long division expand the box below.

Long division explained

Let's suppose we wish to divide 4135 by 17.

```
   0243 r 4
17)4135
   0          0 ×   17 = 0000
   4135    4135 -    0 = 4135
   34      200 ×   17 = 3400
    735    4135 - 3400 =  735
```

```
      68          40 ×    17 =   680
       55        735 -   680 =    55
       51          3 ×    17 =    51
        4         55 -    51 =     4
 Answer: 243 remainder 4
```

First of all we would look at the top digit of the dividend. We see that the smallest multiple of the divisor which is less or equal to it is 0. We output a 0 to the result.

Next we look at the second to top digit of the dividend and all higher digits. We see the smallest multiple of the divisor which is less than or equal is 34. We output a 2 and subtract 3400.

Next we look at the third digit of the dividend and all higher digits. The smallest multiple of the divisor that is less than or equal to this is 68. We output 4 and subtract 680.

Finally we look at all remaining digits. We see that the lowest multiple of the divisor that is less than the remaining digits is 51. We output a 3, subtract 51. The result of the subtraction is our remainder.

To implement division in assembly code, we will implement binary long division. We do this because the numbers are stored in binary, which gives us easy access to the all important bit shift operations, and because division in binary is simpler than in any higher base due to the much lower number of cases.

```
         1011 r 1
1010)1101111
     1010
       11111
       1010
        1011
        1010
           1
```

This example shows how binary long division works. You simply shift the divisor as far right as possible without exceeding the dividend, output a 1 according to the poisition and subtract the number. Whatever remains is the remainder. In this case we show $1101111_2 \div 1010_2 = 1011_2$ remainder 1_2. In decimal, $111 \div 10 = 11$ remainder 1.

Try to implement long division yourself now. You should write a function, DivideU32 which divides r0 by r1, returning the result in r0, and the remainder in r1. Below, we will go through a very efficient implementation.

```
function DivideU32(r0 is dividend, r1 is
divisor)
    set shift to 31
    set result to 0
    while shift ≥ 0
        if dividend ≥ (divisor << shift)
        then
            set dividend to dividend -
            (divisor << shift)
            set result to result + 1
        end if
        set result to result << 1
        set shift to shift - 1
    loop
```

```
    return (result, dividend)
end function
```

This code does achieve what we need, but would not work as assembly code. Our problem comes from the fact that our registers only hold 32 bits, and so the result of divisor << shift may not fit in a register (we call this overflow). This is a real problem. Did your solution have overflow?

Fortunately, an instruction exists called **clz** or count leading zeros, which counts the number of zeros in the binary representation of a number starting at the top bit. Conveniently, this is exactly the number of times we can shift the register left before overflow occurs. Another optimisation you may spot is that we compute divisor << shift twice each loop. We could improve upon this by shifting the divisor at the beginning, then shifting it down at the end of each loop to avoid any need to shift it elsewhere.

Let's have a look at the assembly code to make further improvements.

```
.globl DivideU32
DivideU32:
result .req r0
remainder .req r1
shift .req r2
current .req r3

clz shift,r1
lsl current,r1,shift
mov remainder,r0
mov result,#0
```

```
divideU32Loop$:
	cmp shift,#0
	blt divideU32Return$
	cmp remainder,current

	addge result,result,#1
	subge remainder,current
	sub shift,#1
	lsr current,#1
	lsl result,#1
	b divideU32Loop$

divideU32Return$:
.unreq current
mov pc,lr

.unreq result
.unreq remainder
.unreq shift
```

clz dest,src stores the number of zeros from the top to the first one of register **dest** to register **src**

You may, quite rightly, think that this looks quite efficient. It is pretty good, but division is a very expensive operation, and one we may wish to do quite often, so it would be good if we could improve the speed in any way. When looking to optimise code with a loop in it, it is always important to consider how many times the loop must run. In this case, the loop will run a maximum of 31 times for an input of 1. Without making special cases, this could often be improved easily. For example when dividing 1 by 1, no shift is required, yet we shift the divisor to each of the positions above it. This could be improved by simply using the new **clz**

command on the dividend and subtracting this from the shift. In the case of 1 ÷ 1, this means shift would be set to 0, rightly indicating no shift is required. If this causes the shift to be negative, the divisor is bigger than the dividend and so we know the result is 0 remainder the dividend. Another quick check we could make is if the current value is ever 0, then we have a perfect division and can stop looping.

```
.globl DivideU32
DivideU32:
result .req r0
remainder .req r1
shift .req r2
current .req r3

clz shift,r1
clz r3,r0
subs shift,r3
lsl current,r1,shift
mov remainder,r0
mov result,#0
blt divideU32Return$

divideU32Loop$:
	cmp remainder,current
	blt divideU32LoopContinue$

	add result,result,#1
	subs remainder,current
	lsleq result,shift
	beq divideU32Return$
divideU32LoopContinue$:
	subs shift,#1
	lsrge current,#1
	lslge result,#1
	bge divideU32Loop$
```

```
divideU32Return$:
.unreq current
mov pc,lr

.unreq result
.unreq remainder
.unreq shift
```

Copy the code above to a file called 'maths.s'.

3 Number Strings

Now that we can do division, let's have another look at implementing number to string conversion. The following is pseudo code to convert numbers from registers into strings in up to base 36. By convention, a % b means the remainder of dividing a by b.

```
function SignedString(r0 is value, r1 is
dest, r2 is base)
    if value ≥ 0
    then return UnsignedString(value,
    dest, base)
    otherwise
        if dest > 0 then
            setByte(dest, '-')
            set dest to dest + 1
        end if
        return UnsignedString(-value,
        dest, base) + 1
    end if
end function

function UnsignedString(r0 is value, r1 is
dest, r2 is base)
    set length to 0
    do
        set (value, rem) to
```

```
            DivideU32(value, base)
            if rem > 10
            then set rem to rem + '0'
            otherwise set rem to rem - 10 +
            'a'
            if dest > 0
            then setByte(dest + length, rem)
            set length to length + 1
        while value > 0
        if dest > 0
        then ReverseString(dest, length)
        return length
end function

function ReverseString(r0 is string, r1 is
length)
        set end to string + length - 1
        while end > start
            set temp1 to readByte(start)
            set temp2 to readByte(end)
            setByte(start, temp2)
            setByte(end, temp1)
            set start to start + 1
            set end to end - 1
        end while
end function
```

In a file called 'text.s' implement the above. Remember that if you get stuck, a full solution can be found on the downloads page.

4 Format Strings

Let's get back to our string formatting method. Since we're programming our own operating system, we can add or change formatting rules as we please. We may find it useful to add a %b operation that outputs a

number in binary, and if you're not using null terminated strings, you may wish to alter the behaviour of %s to take the length of the string from another argument, or from a length prefix if you wish. I will use a null terminator in the example below.

One of the main obstacles to implementing this function is that the number of arguments varies. According to the ABI, additional arguments are pushed onto the stack before calling the method in reverse order. So, for example, if we wish to call our method with 8 parameters; 1,2,3,4,5,6,7 and 8, we would do the following:

1. Set r0 = 5, r1 = 6, r2 = 7, r3 = 8
2. Push {r0,r1,r2,r3}
3. Set r0 = 1, r1 = 2, r2 = 3, r3 = 4
4. Call the function
5. Add sp,#4*4

Now we must decide what arguments our function actually needs. In my case, I used the format string address in r0, the length of the format string in r1, the destination string address in r2, followed by the list of arguments required, starting in r3 and continuing on the stack as above. If you wish to use a null terminated format string, the parameter in r1 can be removed. If you wish to have a maximum buffer length, you could store this in r3. As an additional modification, I think it is useful to alter the function so that if the destination string address is 0, no string is outputted, but an accurate length is still returned, so that the length of a formatted string can be accurately determined.

If you wish to attempt the implementation on

your own, try it now. If not, I will first construct the pseudo code for the method, then give the assembly code implementation.

```
function StringFormat(r0 is format, r1 is
formatLength, r2 is dest, ...)
    set index to 0
    set length to 0
    while index < formatLength
        if readByte(format + index) =
        '%' then
            set index to index + 1
            if readByte(format + index)
            = '%' then
                if dest > 0
                then setByte(dest +
                length, '%')
                set length to length + 1
            otherwise if readByte(format
            + index) = 'c' then
                if dest > 0
                then setByte(dest +
                length, nextArg)
                set length to length + 1
            otherwise if readByte(format
            + index) = 'd' or 'i' then
                set length to length +
                SignedString(nextArg,
                dest, 10)
            otherwise if readByte(format
            + index) = 'o' then
                set length to length +
                UnsignedString(nextArg,
                dest, 8)
            otherwise if readByte(format
            + index) = 'u' then
                set length to length +
                UnsignedString(nextArg,
                dest, 10)
            otherwise if readByte(format
```

```
            + index) = 'b' then
                set length to length +
                UnsignedString(nextArg,
                dest, 2)
            otherwise if readByte(format
            + index) = 'x' then
                set length to length +
                UnsignedString(nextArg,
                dest, 16)
            otherwise if readByte(format
            + index) = 's' then
                set str to nextArg
                while getByte(str) !=
                '\0'
                    if dest > 0
                    then setByte(dest +
                    length, getByte(str))
                    set length to length
                    + 1
                    set str to str + 1
                loop
            otherwise if readByte(format
            + index) = 'n' then
                setWord(nextArg,
                length)
            end if
        otherwise
            if dest > 0
            then setByte(dest + length,
            readByte(format + index))
            set length to length + 1
        end if
        set index to index + 1
    loop
    return length
end function
```

Although this function is massive, it is quite straightforward. Most of the code goes into checking all the various conditions, the code

for each one is simple. Further, all the various unsigned integer cases are the same but for the base, and so can be summarised in assembly. This is given below.

```
.globl FormatString
FormatString:
format .req r4
formatLength .req r5
dest .req r6
nextArg .req r7
argList .req r8
length .req r9

push {r4,r5,r6,r7,r8,r9,lr}
mov format,r0
mov formatLength,r1
mov dest,r2
mov nextArg,r3
add argList,sp,#7*4
mov length,#0

formatLoop$:
	subs formatLength,#1
	movlt r0,length
	poplt {r4,r5,r6,r7,r8,r9,pc}

	ldrb r0,[format]
	add format,#1
	teq r0,#'%'
	beq formatArg$

formatChar$:
	teq dest,#0
	strneb r0,[dest]
	addne dest,#1
	add length,#1
	b formatLoop$

formatArg$:
```

```
        subs formatLength,#1
        movlt r0,length
        poplt {r4,r5,r6,r7,r8,r9,pc}

        ldrb r0,[format]
        add format,#1
        teq r0,#'%'
        beq formatChar$

        teq r0,#'c'
        moveq r0,nextArg
        ldreq nextArg,[argList]
        addeq argList,#4
        beq formatChar$

        teq r0,#'s'
        beq formatString$

        teq r0,#'d'
        beq formatSigned$

        teq r0,#'u'
        teqne r0,#'x'
        teqne r0,#'b'
        teqne r0,#'o'
        beq formatUnsigned$

        b formatLoop$

formatString$:
        ldrb r0,[nextArg]
        teq r0,#0x0
        ldreq nextArg,[argList]
        addeq argList,#4
        beq formatLoop$
        add length,#1
        teq dest,#0
        strneb r0,[dest]
        addne dest,#1
        add nextArg,#1
```

```
        b formatString$

formatSigned$:
        mov r0,nextArg
        ldr nextArg,[argList]
        add argList,#4
        mov r1,dest
        mov r2,#10
        bl SignedString
        teq dest,#0
        addne dest,r0
        add length,r0
        b formatLoop$

formatUnsigned$:
        teq r0,#'u'
        moveq r2,#10
        teq r0,#'x'
        moveq r2,#16
        teq r0,#'b'
        moveq r2,#2
        teq r0,#'o'
        moveq r2,#8

        mov r0,nextArg
        ldr nextArg,[argList]
        add argList,#4
        mov r1,dest
        bl UnsignedString
        teq dest,#0
        addne dest,r0
        add length,r0
        b formatLoop$
```

5 Convert OS

Feel free to try using this method however you wish. As an example, here is the code to generate a conversion chart from base 10 to

binary to hexadecimal to octal and to ASCII.

Delete all code after **bl SetGraphicsAddress** in 'main.s' and replace it with the following:

```
mov r4,#0
loop$:
ldr r0, =format
mov r1,#formatEnd-format
ldr r2, =formatEnd
lsr r3,r4,#4
push {r3}
push {r3}
push {r3}
push {r3}
bl FormatString
add sp,#16

mov r1,r0
ldr r0, =formatEnd
mov r2,#0
mov r3,r4

cmp r3,#768-16
subhi r3,#768
addhi r2,#256
cmp r3,#768-16
subhi r3,#768
addhi r2,#256
cmp r3,#768-16
subhi r3,#768
addhi r2,#256

bl DrawString

add r4,#16
b loop$

.section .data
format:
```

```
.ascii "%d = 0b%b = 0x%x = 0%o = '%c'"
formatEnd:
```

Can you work out what will happen before testing? Particularly what happens for r3 ≥ 128? Try it on the Raspberry Pi to see if you're right. If it doesn't work, please see our troubleshooting page.

When it does work, congratulations, you've completed the Screen04 tutorial, and reached the end of the screen series! We've learned about pixels and frame buffers, and how these apply to the Raspberry Pi. We've learned how to draw simple lines, and also how to draw characters, as well as the invaluable skill of formatting numbers into text. We now have all that you would need to make graphical output on an Operating System. Can you make some more drawing methods? What about 3D graphics? Can you implement a 24bit frame buffer? What about reading the size of the framebuffer in from the command line?

The next series is the Input series, which teaches how to use the keyboard and mouse to really get towards a traditional console computer.

Spot a mistake? You can help improve this tutorial on GitHub.

Based on contributions at https://github.com/chadderz121/bakingpi-www.

Lesson 10 Input01

Welcome to the Input lesson series. In this series, you will learn how to receive inputs to the Raspberry Pi using the keyboard. We will start with just revealing the input, and then move to a more traditional text prompt.

This first input lesson teaches some theory about drivers and linking, as well as about keyboards and ends up displaying text on the screen.

1 Getting Started

It is expected that you have completed the OK series, and it would be helpful to have completed the Screen series. Many of the files from that series will be called, without comment. If you do not have these files, or prefer to use a correct implementation, download the template for this lesson from the Downloads page. If you're using your own implementation, please remove everything after your call to SetGraphicsAddress.

2 USB

> The USB standard was designed to make simple hardware in exchange for complex software.

As you are no doubt aware, the Raspberry Pi model B has two USB ports, commonly used for connecting a mouse and keyboard. This was a very good design decision, USB is a

very generic connector, and many different kinds of device use it. It's simple to build new devices for, simple to write device drivers for, and is highly extensible thanks to USB hubs. Could it get any better? Well, no, in fact for an Operating Systems developer this is our worst nightmare. The USB standard is huge. I really mean it this time, it is over 700 pages, before you've even thought about connecting a device.

I spoke to a number of other hobbyist Operating Systems developers about this and they all say one thing: don't bother. "It will take too long to implement", "You won't be able to write a tutorial on it" and "It will be of little benefit". In many ways they are right, I'm not able to write a tutorial on the USB standard, as it would take weeks. I also can't teach how to write device drivers for all the different devices, so it is useless on its own. However, I can do the next best thing: Get a working USB driver, get a keyboard driver, and then teach how to use these in an Operating System. I set out searching for a free driver that would run in an operating system that doesn't even know what a file is yet, but I couldn't find one. They were all too high level. So, I attempted to write one. Everybody was right, this took weeks to do. However, I'm pleased to say I did get one that works with no external help from the Operating System, and can talk to a mouse and keyboard. It is by no means complete, efficient, or correct, but it does work. It has been written in C and the full source code can be found on the downloads page for those interested.

So, this tutorial won't be a lesson on the USB

standard (at all). Instead we'll look at how to work with other people's code.

3 Linking

> Linking allows us to make reusable code 'libraries' that anyone can use in their program.

Since we're about to incorporate external code into the Operating System, we need to talk about linking. Linking is a process which is applied to programs or Operating System to link in functions. What this means is that when a program is made, we don't necessarily code every function (almost certainly not in fact). Linking is what we do to make our program link to functions in other people's code. This has actually been going on all along in our Operating Systems, as the linker links together all of the different files, each of which is compiled separately.

> Programs often just call libraries, which call other libraries and so on until eventually they call an Operating System library which we would write.

There are two types of linking: static and dynamic. Static linking is like what goes on when we make our Operating Systems. The linker finds all the addresses of the functions, and writes them into the code, before the program is finished. Dynamic linking is linking that occurs after the program is 'complete'. When it is loaded, the dynamic linker goes through the program and links any functions which are not in the program to libraries in the Operating System. This is

one of the jobs our Operating System should eventually be capable of, but for now everything will be statically linked.

The USB driver I have written is suitable for static linking. This means I give you the compiled code for each of my files, and then the linker finds functions in your code which are not defined in your code, and links them to functions in my code. On the Downloads page for this lesson is a makefile and my USB driver, which you will need to continue. Download them and replace the makefile in your code with this one, and also put the driver in the same folder as that makefile.

4 Keyboards

In order to get input into our Operating System, we need to understand at some level how keyboards actually work. Keyboards have two types of keys: Normal and Modifier keys. The normal keys are the letters, numbers, function keys, etc. They constitute almost every key on the keyboard. The modifiers are up to 8 special keys. These are left shift, right shift, left control, right control, left alt, right alt, left GUI and right GUI. The keyboard can detect any combination of the modifier keys being held, as well as up to 6 normal keys. Every time a key changes (i.e. is pushed or released), it reports this to the computer. Typically, keyboards also have three LEDs for Caps Lock, Num Lock and Scroll Lock, which are controlled by the computer, not the keyboard itself. Keyboards may have many more lights such as power, mute, etc.

In order to help standardise USB keyboards, a

table of values was produced, such that every keyboard key ever is given a unique number, as well as every conceivable LED. The table below lists the first 126 of values.

Number	Description	Number	Description	Number	Description	Number	Description
4	a and A	5	b and B	6	c and C	7	d and D
8	e and E	9	f and F	10	g and G	11	h and H
12	i and I	13	j and J	14	k and K	15	l and L
16	m and M	17	n and N	18	o and O	19	p and P
20	q and Q	21	r and R	22	s and S	23	t and T
24	u and U	25	v and V	26	w and W	27	x and X
28	y and Y	29	z and Z	30	1 and !	31	2 and @
32	3 and #	33	4 and $	34	5 and %	35	6 and ^
36	7 and &	37	8 and *	38	9 and (	39	0 and)
40	Return (Enter)	41	Escape	42	Delete (Backspace)	43	Tab
44	Spacebar	45	- and _	46	= and +	47	[and {
48	] and }	49	\ and |	50	# and ~	51	; and :
52	' and "	53	` and ~	54	, and <	55	. and >
56	/ and ?	57	Caps Lock	58	F1	59	F2
60	F3	61	F4	62	F5	63	F6
64	F7	65	F8	66	F9	67	F10

68	F11	69	F12	70	Print Screen	71	Scroll Lock
72	Pause	73	Insert	74	Home	75	Page Up
76	Delete forward	77	End	78	Page Down	79	Right Arrow
80	Left Arrow	81	Down Arrow	82	Up Arrow	83	Num Lock
84	Keypad /	85	Keypad *	86	Keypad -	87	Keypad +
88	Keypad Enter	89	Keypad 1 and End	90	Keypad 2 and Down Arrow	91	Keypad 3 and Page Down
92	Keypad 4 and Left Arrow	93	Keypad 5	94	Keypad 6 and Right Arrow	95	Keypad 7 and Home
96	Keypad 8 and Up Arrow	97	Keypad 9 and Page Up	98	Keypad 0 and Insert	99	Keypad . and Delete
100	\ and |	101	Application	102	Power	103	Keypad =
104	F13	105	F14	106	F15	107	F16
108	F17	109	F18	110	F19	111	F20
112	F21	113	F22	114	F23	115	F24
116	Execute	117	Help	118	Menu	119	Select
120	Stop	121	Again	122	Undo	123	Cut
124	Copy	125	Paste	126	Find	127	Mute
128	Volume Up	129	Volume Down				

Table 4.1 USB Keyboard Keys

The full list can be found in section 10, page 53 of HID Usage Tables 1.12.

5 The Nut Behind the Wheel

These summaries and the code they describe form an API - Application Product Interface.

Normally, when you work with someone else's code, they provide a summary of their methods, what they do and roughly how they work, as well as how they can go wrong. Here is a table of the relevant instructions required to use my USB driver.

Function	Arguments	Returns	Description
UsbInitialise	None	r0 is result code	This method is the all-in-one method that loads the USB driver, enumerates all devices and attempts to communicate with them. This method generally takes about a second to execute, though with a few USB hubs plugged in this can be significantly longer. After this method is

			completed methods in the keyboard driver become available, regardless of whether or not a keyboard is indeed inserted. Result code explained below.
UsbCheckForChange	No	None	Essentially provides the same effect as UsbInitialise, but does not provide the same one time initialisation. This method checks every port on every connected hub recursively, and adds new devices if they have

			been added. This should be very quick if there are no changes, but can take up to a few seconds if a hub with many devices is attached.
KeyboardCount	None	r0 is count	Returns the number of keyboards currently connected and detected. UsbCheckForChange may update this. Up to 4 keyboards are supported by default. Up to this many keyboards may be accessed through this driver.

KeyboardGetAddress	r0 is index	r0 is address	Retrieves the address of a given keyboard. All other functions take a keyboard address in order to know which keyboard to access. Thus, to communicate with a keyboard, first check the count, then retrieve the address, then use other methods. Note, the order of keyboards that this method returns may change after calls to UsbCheckForChange.
KeyboardPoll	r0 is address	r0 is result code	Reads in the current

			key state from the keyboard. This operates via polling the device directly, contrary to the best practice. This means that if this method is not called frequently enough, a key press could be missed. All reading methods simply return the value as of the last poll.
KeyboardGetModifiers	r0 is address	r0 is modifier state	Retrieves the status of the modifier keys as of the last poll. These are the shift, alt control and GUI keys on both sides. This

			is returned as a bit field, such that a 1 in the bit 0 means left control is held, bit 1 means left shift, bit 2 means left alt, bit 3 means left GUI and bits 4 to 7 mean the right versions of those previous. If there is a problem r0 contains 0.
KeyboardGetKeyDownCount	r0: address	C0: int count	Retrieves the number of keys currently held down on the keyboard. This excludes modifier keys. Normally, this cannot go above 6. If there is an error

			this method returns 0.
KeyboardGetKeyDown	r0 is address, r1 code is key number		Retrieves the scan code (see Table 4.1) of a particular held down key. Normally, to work out which keys are down, call KeyboardGetKeyDownCount and then call KeyboardGetKeyDown up to that many times with increasing values of r1 to determine which keys are down. Returns 0 if there is a problem. It is safe (but not recommended) to call this method without calling KeyboardGetKeyDownCount

			and interpret 0s as keys not held. Note, the order or scan codes can change randomly (some keyboards sort numerically, some sort temporally, no guarantees are made).
KeyboardGetKeyIsDown	r0 is address, r1 is scan code	r0 is status	Alternative to KeyboardGetKeyDown, checks if a particular scan code is among the held down keys. Returns 0 if not, or a non-zero value if so. Faster when detecting particular scan codes (e.g. looking for ctrl + c). On error,

KeyboardGetLedSupport	0 is reserved, 1 is address	0 is LEDs	returns 0. Checks which LEDs a particular keyboard supports. Bit 0 being 1 represents Number Lock, bit 1 represents Caps Lock, bit 2 represents Scroll Lock, bit 3 represents Compose, bit 4 represents Kana, bit 5 represents Power, bit 6 represents Mute and bit 7 represents Compose. As per the USB standard, none of these LEDs update automatically (e.g. Caps Lock must

			be set manually when the Caps Lock scan code is detected).
KeyboardSetLeds	r0 is address, r1 is LEDs	r0 is result code	Attempts to turn on/off the specified LEDs on the keyboard. See below for result code values. See KeyboardGetLedSupport for LEDs' values.

Table 5.1 Keyboard related functions in CSUD

> Result codes are an easy way to handle errors, but often more elegant solutions exist in higher level code.

Several methods return 'result codes'. These are commonplace in C code, and are just numbers which represent what happened in a method call. By convention, 0 always indicates success. The following result codes are used by this driver.

Code	Description
0	Method completed successfully.
-2	Argument: A method was called with an invalid argument.

-4	Device: The device did not respond correctly to the request.
-5	Incompatible: The driver is not compatible with this request or device.
-6	Compiler: The driver was compiled incorrectly, and is broken.
-7	Memory: The driver ran out of memory.
-8	Timeout: The device did not respond in the expected time.
-9	Disconnect: The device requested has disconnected, and cannot be used.

Table 5.2 - CSUD Result Codes

The general usage of the driver is as follows:

1. Call UsbInitialise
2. Call UsbCheckForChange
3. Call KeyboardCount
4. If this is 0, go to 2.
5. For each keyboard you support:
    1. Call KeyboardGetAddress
    2. Call KeybordGetKeyDownCount
    3. For each key down:
        1. Check whether or not it has just been pushed
        2. Store that the key is down
    4. For each key stored:
        1. Check whether or not key

is released
2. Remove key if released

6. Perform actions based on keys pushed/ released
7. Go to 2.

Ultimately, you may do whatever you wish to with the keyboard, and these methods should allow you to access all of its functionality. Over the next 2 lessons, we shall look at completing the input side of a text terminal, similarly to most command line computers, and interpreting the commands. In order to do this, we're going to need to have keyboard inputs in a more useful form. You may notice that my driver is (deliberately) unhelpful, because it doesn't have methods to deduce whether or not a key has just been pushed down or released, it only has methods about what is currently held down. This means we'll need to write such methods ourselves.

6 Updates Available

> Repeatedly checking for updates is called 'polling'. This is in contrast to interrupt driven IO, where the device sends a signal when data is ready.

First of all, let's implement a method KeyboardUpdate which detects the first keyboard and uses its poll method to get the current input, as well as saving the last inputs for comparison. We can then use this data with other methods to translate scan codes to keys. The method should do precisely the following:

1. Retrieve a stored keyboard address

(initially 0).
2. If this is not 0, go to 9.
3. Call UsbCheckForChange to detect new keyboards.
4. Call KeyboardCount to detect how many keyboards are present.
5. If this is 0 store the address as 0 and return; we can't do anything with no keyboard.
6. Call KeyboardGetAddress with parameter 0 to get the first keyboard's address.
7. Store this address.
8. If this is 0, return; there is some problem.
9. Call KeyboardGetKeyDown 6 times to get each key currently down and store them
10. Call KeyboardPoll
11. If the result is non-zero go to 3. There is some problem (such as disconnected keyboard).

To store the values mentioned above, we will need the following values in the .data section.

```
.section .data
.align 2
KeyboardAddress:
.int 0
KeyboardOldDown:
.rept 6
.hword 0
.endr
```

.hword num inserts the half word constant **num** into the file directly.

.rept num [commands] .endr copies the commands **commands** to the output **num** times.

Try to implement the method yourself. My implementation for this is as follows:

1. ```
   .section .text
   .globl KeyboardUpdate
   KeyboardUpdate:
   push {r4,r5,lr}

   kbd .req r4
   ldr r0,=KeyboardAddress
   ldr kbd,[r0]
   ```
   We load in the keyboard address.
2. ```
   teq kbd,#0
   bne haveKeyboard$
   ```
   If the address is non-zero, we have a keyboard. Calling UsbCheckForChanges is slow, and so if everything works we avoid it.
3. ```
   getKeyboard$:
   bl UsbCheckForChange
   ```
   If we don't have a keyboard, we have to check for new devices.
4. ```
   bl KeyboardCount
   ```
   Now we see if a new keyboard has been added.
5. ```
   teq r0,#0
   ldreq r1,=KeyboardAddress
   streq r0,[r1]
   beq return$
   ```
   There are no keyboards, so we have no keyboard address.

```
```
```

6.
```
mov r0,#0
bl KeyboardGetAddress
```
Let's just get the address of the first keyboard. You may want to allow more.

7.
```
ldr r1,=KeyboardAddress
str r0,[r1]
```
Store the keyboard's address.

8.
```
teq r0,#0
beq return$
mov kbd,r0
```
If we have no address, there is nothing more to do.

9.
```
saveKeys$:
        mov r0,kbd
        mov r1,r5
        bl KeyboardGetKeyDown

        ldr r1,=KeyboardOldDown
        add r1,r5,lsl #1
        strh r0,[r1]
        add r5,#1
        cmp r5,#6
        blt saveKeys$
```
Loop through all the keys, storing them in KeyboardOldDown. If we ask for too many, this returns 0 which is fine.

10.
```
mov r0,kbd
bl KeyboardPoll
```
Now we get the new keys.

11.
```
teq r0,#0
bne getKeyboard$

return$:
```

```
pop {r4,r5,pc}
.unreq kbd
```

Finally we check if KeyboardPoll worked. If not, we probably disconnected.

With our new KeyboardUpdate method, checking for inputs becomes as simple as calling this method at regular intervals, and it will even check for disconnections etc. This is a useful method to have, as our actual key processing may differ based on the situation, and so being able to get the current input in its raw form with one method call is generally applicable. The next method we ideally want is KeyboardGetChar, a method that simply returns the next key pressed as an ASCII character, or returns 0 if no key has just been pressed. This could be extended to support typing a key multiple times if it is held for a certain duration, and to support the 'lock' keys as well as modifiers.

To make this method it is useful if we have a method KeyWasDown, which simply returns 0 if a given scan code is not in the KeyboardOldDown values, and returns a non-zero value otherwise. Have a go at implementing this yourself. As always, a solution can be found on the downloads page.

7 Look Up Tables

In many areas of programming, the larger the program, the faster it is. Look up tables are large, but are very fast. Some problems can be solved by a mixture of look up tables and normal functions.

The KeyboardGetChar method could be quite complex if we write it poorly. There are 100s of scan codes, each with different effects depending on the presence or absence of the shift key or other modifiers. Not all of the keys can be translated to a character. For some characters, multiple keys can produce the same character. A useful trick in situations with such vast arrays of possibilities is look up tables. A look up table, much like in the physical sense, is a table of values and their results. For some limited functions, the simplest way to deduce the answer is just to precompute every answer, and just return the correct one by retrieving it. In this case, we could build up a sequence of values in memory such that the nth value into the sequence is the ASCII character code for the scan code n. This means our method would simply have to detect if a key was pressed, and then retrieve its value from the table. Further, we could have a separate table for the values when shift is held, so that the shift key simply changes which table we're working with.

After the **.section .data** command, copy the following tables:

```
.align 3
KeysNormal:
	.byte 0x0, 0x0, 0x0, 0x0, 'a', 'b', 'c', 'd'
	.byte 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l'
	.byte 'm', 'n', 'o', 'p', 'q', 'r', 's', 't'
	.byte 'u', 'v', 'w', 'x', 'y', 'z', '1', '2'
	.byte '3', '4', '5', '6', '7', '8', '9', '0'
	.byte '\n', 0x0, '\b', '\t', ' ', '-', '=', '['
	.byte ']', '\\', '#', ';', '\"', '`', ',', '.'
	.byte '/', 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
	0x0
```

```
        .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
        0x0, 0x0
        .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
        0x0, 0x0
        .byte 0x0, 0x0, 0x0, 0x0, '/', '*', '-', '+'
        .byte '\n', '1', '2', '3', '4', '5', '6', '7'
        .byte '8', '9', '0', '.', '\\', 0x0, 0x0, '='

.align 3
KeysShift:
        .byte 0x0, 0x0, 0x0, 0x0, 'A', 'B', 'C',
        'D'
        .byte 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L'
        .byte 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T'
        .byte 'U', 'V', 'W', 'X', 'Y', 'Z', '!', ""
        .byte '£', '$', '%', '^', '&', '*', '(', ')'
        .byte '\n', 0x0, '\b', '\t', ' ', '_', '+', '{'
        .byte '}', '|', '~', ':', '@', '¬', '<', '>'
        .byte '?', 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
        0x0
        .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
        0x0, 0x0
        .byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
        0x0, 0x0
        .byte 0x0, 0x0, 0x0, 0x0, '/', '*', '-', '+'
        .byte '\n', '1', '2', '3', '4', '5', '6', '7'
        .byte '8', '9', '0', '.', '|', 0x0, 0x0, '='
```

.byte num inserts the byte constant **num** into the file directly.

Most assemblers and compilers recognise escape sequences; character sequences such as \t which insert special characters instead.

These tables map directly the first 104 scan codes onto the ASCII characters as a table of bytes. We also have a separate table describing the effects of the shift key on

those scan codes. I've used the ASCII null character (0) for all keys without direct mappings in ASCII (such as the function keys). Backspace is mapped to the ASCII backspace character (8 denoted \b), enter is mapped to the ASCII new line character (10 denoted \n) and tab is mapped to the ASCII horizontal tab character (9 denoted \t).

The KeyboardGetChar method will need to do the following:

1. Check if KeyboardAddress is 0. If so, return 0.
2. Call KeyboardGetKeyDown up to 6 times. Each time:
    1. If key is 0, exit loop.
    2. Call KeyWasDown. If it was, go to the next key.
    3. If the scan code is more than 103, go to the next key.
    4. Call KeyboardGetModifiers
    5. If shift is held, load the address of KeysShift. Otherwise load KeysNormal.
    6. Read the ASCII value from the table.
    7. If it is 0, go to the next key otherwise return this ASCII code and exit.
3. Return 0.

Try to implement this yourself. My implementation is presented below:

1.
```
.globl KeyboardGetChar
KeyboardGetChar:
ldr r0, =KeyboardAddress
```

```
ldr r1,[r0]
teq r1,#0
moveq r0,#0
moveq pc,lr
```

Simple check to see if we have a keyboard.

2.
```
push {r4,r5,r6,lr}
kbd .req r4
key .req r6
mov r4,r1
mov r5,#0
keyLoop$:
    mov r0,kbd
    mov r1,r5
    bl KeyboardGetKeyDown
```

r5 will hold the index of the key, r4 holds the keyboard address.

1.
```
teq r0,#0
beq keyLoopBreak$
```

If a scan code is 0, it either means there is an error, or there are no more keys.

2.
```
mov key,r0
bl KeyWasDown
teq r0,#0
bne keyLoopContinue$
```

If a key was already down it is uninteresting, we only want ot know about key presses.

```
```

3.
```
cmp key,#104
bge keyLoopContinue$
```

If a key has a scan code higher than 104, it will be outside our table, and so is not relevant.

4.
```
mov r0,kbd
bl KeyboardGetModifiers
```

We need to know about the modifier keys in order to deduce the character.

5.
```
tst r0,#0b00100010
ldreq r0,=KeysNormal
ldrne r0,=KeysShift
```

We detect both a left and right shift key as changing the characters to their shift variants. Remember, a **tst** instruction computes the logical AND and then compares it to zero, so it will be equal to 0 if and only if both of the shift bits are zero.

6.
```
ldrb r0,[r0,key]
```

Now we can load in the key from the look up table.

7.
```
teq r0,#0
bne keyboardGetCharReturn$
keyLoopContinue$:
add r5,#1
cmp r5,#6
```

```
blt keyLoop$
```

If the look up code contains a zero, we must continue. To continue, we increment the index, and check if we've reached 6.

3.
```
keyLoopBreak$:
mov r0,#0
keyboardGetCharReturn$:
pop {r4,r5,r6,pc}
.unreq kbd
.unreq key
```

We return our key here, if we reach keyLoopBreak$, then we know there is no key held, so return 0.

8 Notepad OS

Now we have our KeyboardGetChar method, we can make an operating system that just types what the user writes to the screen. For simplicity we'll ignore all the unusual keys. In 'main.s' delete all code after **bl SetGraphicsAddress**. Call UsbInitialise, set r4 and r5 to 0, then loop forever over the following commands:

1. Call KeyboardUpdate
2. Call KeyboardGetChar
3. If it is 0, got to 1
4. Copy r4 and r5 to r1 and r2 then call DrawCharacter
5. Add r0 to r4
6. If r4 is 1024, add r1 to r5 and set r4 to 0

7. If r5 is 768 set r5 to 0
8. Go to 1

Now compile this and test it on the Pi. You should almost immediately be able to start typing text to the screen when the Pi starts. If not, please see our troubleshooting page.

When it works, congratulations, you've achieved an interface with the computer. You should now begin to realise that you've almost got a primitive operating system together. You can now interface with the computer, issuing it commands, and receive feedback on screen. In the next tutorial, Input02 we will look at producing a full text terminal, in which the user types commands, and the computer executes them.

Based on contributions at https://github.com/chadderz121/bakingpi-www.

Lesson 11 Input02

The Input02 lesson builds on Input01, by building a simple command line interface where the user can type commands and the computer interprets and displays them. It is assumed you have the code for the Lesson 11: Input01 operating system as a basis.

1 Terminal 1

> In the early days of computing, there would usually be one large computer in a building, and many 'terminals' which sent commands to it. The computer would take it in turns to execute different incoming commands.

Almost every operating system starts life out as a text terminal. This is typically a black screen with white writing, where you type commands for the computer to execute on the keyboard, and it explains how you've mistyped them, or very occasionally, does what you want. This approach has two main advantages: it provides a simple, robust control mechanism for the computer using only a keyboard and monitor, and it is done by almost every operating system, so is widely understood by system administrators.

Let's analyse what we want to do precisely:

1. Computer turns on, displays some sort of welcome message
2. Computer indicates its ready for input
3. User types a command, with parameters, on the keyboard

4. User presses return or enter to commit the command
5. Computer interprets command and performs actions if command is acceptable
6. Computer displays messages to indicate if command was successful, and also what happened
7. Loop back to 2

One defining feature of such terminals is that they are unified for both input and output. The same screen is used to enter inputs as is used to print outputs. This means it is useful to build an abstraction of a character based display. In a character based display, the smallest unit is a character, not a pixel. The screen is divided into a fixed number of characters which have varying colours. We can build this on top of our existing screen code, by storing the characters and their colours, and then using the DrawCharacter method to push them to the screen. Once we have a character based display, drawing text becomes a matter of drawing a line of characters.

In a new file called terminal.s copy the following code:

```
.section .data
.align 4
terminalStart:
.int terminalBuffer
terminalStop:
.int terminalBuffer
terminalView:
.int terminalBuffer
terminalColour:
.byte 0xf
```

```
.align 8
terminalBuffer:
.rept 128*128
.byte 0x7f
.byte 0x0
.endr
terminalScreen:
.rept 1024/8 * 768/16
.byte 0x7f
.byte 0x0
.endr
```

This sets up the data we need for the text terminal. We have two main storages: terminalBuffer and terminalScreen. terminalBuffer is storage for all of the text we have displayed. It stores up to 128 lines of text (each containing 128 characters). Each character consists of an ASCII character code and a colour, all of which are initially set to 0x7f (ASCII delete) and 0 (black on a black background). terminalScreen stores the characters that are currently displayed on the screen. It is 128 by 48 characters, similarly initialised. You may think that we only need this terminalScreen, not the terminalBuffer, but storing the buffer has 2 main advantages:

1. We can easily see which characters are different, so we only have to draw those.
2. We can 'scroll' back through the terminal's history because it is stored (to a limit).

You should always try to design systems that do the minimum amount of work, as they run much faster for things which don't often change.

The differing trick is really common on low power Operating Systems. Drawing the screen is a slow operation, and so we only want to draw thing that we absolutely have to. In this system, we can freely alter the terminalBuffer, and then call a method which copies the bits that change to the screen. This means we don't have to draw each character as we go along, which may save time in the long run on very long sections of text that span many lines.

The other values in the .data section are as follows:

terminalStart
: The first character which has been written in terminalBuffer.

terminalStop
: The last character which has been written in terminalBuffer.

terminalView
: The first character on the screen at present. We can use this to scroll the screen.

temrinalColour
: The colour to draw new characters with.

Circular buffers are an example of an **data structure**. These are just ideas we have for organising data, that we sometimes implement in software.

The reason why terminalStart needs to be stored is because termainlBuffer should be a circular buffer. This means that when the buffer is completely full, the end 'wraps' round to the start, and so the character after the very last one is the first one. Thus, we need to advance terminalStart so we know that we've done this. When wokring with the buffer this can easily be implemented by checking if the index goes beyond the end of the buffer, and setting it back to the beginning if it does. Circular buffers are a common and clever way of storing a lot of data, where only the most recent data is important. It allows us to keep writing indefinitely, while always being sure there is a certain amount of recent data available. They're often used in signal processing or compression algorithms. In this case, it allows us to store a 128 line history of the terminal, without any penalties for writing over 128 lines. If we didn't have this, we would have to copy 127 lines back a line very time we went beyond the 128th line, wasting valuable time.

I've mentioned the terminalColour here a few times. You can implement this however you, wish, however there is something of a standard on text terminals to have only 16 colours for foreground, and 16 colours for

background (meaning there are $16^2 = 256$ combinations). The colours on a CGA terminal are defined as follows:

Number	Colour (R, G, B)
0	Black (0, 0, 0)
1	Blue (0, 0, ⅔)
2	Green (0, ⅔, 0)
3	Cyan (0, ⅔, ⅔)
4	Red (⅔, 0, 0)
5	Magenta (⅔, 0, ⅔)
6	Brown (⅔, ⅓, 0)
7	Light Grey (⅔, ⅔, ⅔)
8	Grey (⅓, ⅓, ⅓)
9	Light Blue (⅓, ⅓, 1)
10	Light Green (⅓, 1, ⅓)
11	Light Cyan (⅓, 1, 1)
12	Light Red (1, ⅓, ⅓)
13	Light Magenta (1, ⅓, 1)
14	Yellow (1, 1, ⅓)
15	White (1, 1, 1)

Table 1.1 - CGA Colour Codes

> Brown was used as the alternative (dark yellow) was unappealing and not useful.

We store the colour of each character by storing the fore colour in the low nibble of the colour byte, and the background colour in the high nibble. Apart from brown, all of these colours follow a pattern such that in binary, the top bit represents adding ⅓ to each component, and the other bits represent adding ⅔ to individual components. This makes it easy to convert to RGB colour values.

We need a method, TerminalColour, to read these 4 bit colour codes, and then call

SetForeColour with the 16 bit equivalent. Try to implement this on your own. If you get stuck, or have not completed the Screen series, my implementation is given below:

```
.section .text
TerminalColour:
teq r0,#6
ldreq r0,=0x02B5
beq SetForeColour

tst r0,#0b1000
ldrne r1,=0x52AA
moveq r1,#0
tst r0,#0b0100
addne r1,#0x15
tst r0,#0b0010
addne r1,#0x540
tst r0,#0b0001
addne r1,#0xA800
mov r0,r1
b SetForeColour
```

2 Showing the Text

The first method we really need for our terminal is TerminalDisplay, one that copies the current data from terminalBuffer to terminalScreen and the actual screen. As mentioned, this method should do a minimal amount of work, because we need to be able to call it often. It should compare the text in terminalBuffer with that in terminalDisplay, and copy it across if they're different. Remember, terminalBuffer is a circular buffer running, in this case, from terminalView to terminalStop or 128*48 characters, whichever comes sooner. If we hit

terminalStop, we'll assume all characters after that point are $7f_{16}$ (ASCII delete), and have colour 0 (black on a black background).

Let's look at what we have to do:

1. Load in terminalView, terminalStop and the address of terminalDisplay.
2. For each row:
    1. For each column:
        1. If view is not equal to stop, load the current character and colour from view
        2. Otherwise load the character as 0x7f and the colour as 0
        3. Load the current character from terminalDisplay
        4. If the character and colour are equal, go to 10
        5. Store the character and colour to terminalDisplay
        6. Call TerminalColour with the background colour in r0
        7. Call DrawCharacter with r0 = 0x7f (ASCII delete, a block), r1 = x, r2 = y
        8. Call TerminalColour with the foreground colour in r0
        9. Call DrawCharacter with r0 = character, r1 = x, r2 = y
        10. Increment the position in terminalDisplay by 2
        11. If view and stop are not equal, increment the view position by 2
        12. If the view position is at

the end of textBuffer, set it to the start

13. Increment the x co-ordinate by 8

2. Increment the y co-ordinate by 16

Try to implement this yourself. If you get stuck, my solution is given below:

1.
```
.globl TerminalDisplay
TerminalDisplay:
push {r4,r5,r6,r7,r8,r9,r10,r11,lr}
x .req r4
y .req r5
char .req r6
col .req r7
screen .req r8
taddr .req r9
view .req r10
stop .req r11

ldr taddr,=terminalStart
ldr view,[taddr,#terminalView -
terminalStart]
ldr stop,[taddr,#terminalStop -
terminalStart]
add taddr,#terminalBuffer -
terminalStart
add taddr,#128*128*2
mov screen,taddr
```

I go a little wild with variables here. I'm using taddr to store the location of the end of the textBuffer for ease.

2.
```
mov y,#0
yLoop$:
```

Start off the y loop.

1.
```
mov x,#0
xLoop$:
```

   Start off the x loop.

   1.
```
teq view,stop
ldrneh char,[view]
```

      I load both the character and the colour into char simultaneously for ease.

   2.
```
moveq char,#0x7f
```

      This line complements the one above by acting as though a black delete character was read.

   3.
```
ldrh col,[screen]
```

      For simplicity I load both the character and colour into col simultaneously.

   4.
```
teq col,char
beq xLoopContinue$
```

      Now we can check if anything has changed with a teq.

   5.
```
strh char,[screen]
```

We can also easily save the current value.

6.
```
lsr col,char,#8
and char,#0x7f
lsr r0,col,#4
bl TerminalColour
```

I split up char into the colour in col and the character in char with a bitshift and an and, then use a bitshift to get the background colour to call TerminalColour.

7.
```
mov r0,#0x7f
mov r1,x
mov r2,y
bl DrawCharacter
```

Write out a delete character which is a coloured block.

8.
```
and r0,col,#0xf
bl TerminalColour
```

Use an and to get the low nibble of col then call TerminalColour.

9.
```
mov r0,char
mov r1,x
mov r2,y
bl DrawCharacter
```

Write out the character we're supposed to write.

10.
```
xLoopContinue$:
add screen,#2
```

Increment the screen pointer.

11.
```
teq view,stop
addne view,#2
```

Increment the view pointer if necessary.

12.
```
teq view,taddr
subeq
view,#128*128*2
```

It's easy to check for view going past the end of the buffer because the end of the buffer's address is stored in taddr.

13.
```
add x,#8
teq x,#1024
bne xLoop$
```

We increment x and then loop back if there are more characters to go.

2.
```
add y,#16
teq y,#768
bne yLoop$
```

We increment y and then loop back if there are more characters to go.

```
pop {r4,r5,r6,r7,r8,r9,r10,r11,pc}
.unreq x
.unreq y
.unreq char
.unreq col
.unreq screen
.unreq taddr
.unreq view
.unreq stop
```

Don't forget to clean up at the end!

3 Printing Lines

Now we have our TerminalDisplay method, which will automatically display the contents of terminalBuffer to terminalScreen, so theoretically we can draw text. However, we don't actually have any drawing routines that work on a character based display. A quick method that will come in handy first of all is TerminalClear, which completely clears the terminal. This can actually very easily be achieved with no loops. Try to deduce why the following method suffices:

```
.globl TerminalClear
TerminalClear:
ldr r0,=terminalStart
add r1,r0,#terminalBuffer-terminalStart
str r1,[r0]
str r1,[r0,#terminalStop-terminalStart]
str r1,[r0,#terminalView-terminalStart]
mov pc,lr
```

Now we need to make a basic method for character based displays; the Print function. This takes in a string address in r0, and a length in r1, and simply writes it to the current location at the screen. There are a few special characters to be wary of, as well as special behaviour to ensure that terminalView is kept up to date. Let's analyse what it has to do:

1. Check if string length is 0, if so return
2. Load in terminalStop and terminalView
3. Deduce the x-coordinate of terminalStop
4. For each character:
    1. Check if the character is a new line
    2. If so, increment bufferStop to the end of the line storing a black on black delete character.
    3. Otherwise, copy the character in the current terminalColour
    4. Check if we're at the end of a line
    5. If so, check if the number of characters between terminalView and terminalStop is more than one screen
    6. If so, increment terminalView by one line
    7. Check if terminalView is at the end of the buffer, replace it with the start if so
    8. Check if terminalStop is at the end of the buffer, replace it with the start if so
    9. Check if terminalStop equals terminalStart, increment terminalStart by one line if so
    10. Check if terminalStart is at the

end of the buffer, replace it with the start if so

5. Store back terminalStop and terminalView.

See if you can implement this yourself. My solution is provided below:

1.
```
.globl Print
Print:
teq r1,#0
moveq pc,lr
```

This quick check at the beginning makes a call to Print with a string of length 0 almost instant.

2.
```
push {r4,r5,r6,r7,r8,r9,r10,r11,lr}
bufferStart .req r4
taddr .req r5
x .req r6
string .req r7
length .req r8
char .req r9
bufferStop .req r10
view .req r11

mov string,r0
mov length,r1

ldr taddr, =terminalStart
ldr bufferStop,
[taddr,#terminalStop-terminalStart]
ldr view,[taddr,#terminalView-
terminalStart]
ldr bufferStart,[taddr]
add taddr,#terminalBuffer-
terminalStart
```

```
add taddr,#128*128*2
```

I do a lot of setup here. bufferStart contains terminalStart, bufferStop contains terminalStop, view contains terminalView, taddr is the address of the end of terminalBuffer.

3.
```
and x,bufferStop,#0xfe
lsr x,#1
```

As per usual, a sneaky alignment trick makes everything easier. Because of the aligment of terminalBuffer, the x-coordinate of any character address is simply the last 8 bits divided by 2.

4.
    1.
    ```
    charLoop$:
    ldrb char,[string]
    and char,#0x7f
    teq char,#'\n'
    bne charNormal$
    ```

    We need to check for new lines.

    2.
    ```
    mov r0,#0x7f
    clearLine$:
    strh r0,[bufferStop]
    add bufferStop,#2
    add x,#1
    teq x,#128 blt clearLine$

    b charLoopContinue$
    ```

    Loop until the end of the line, writing out 0x7f; a delete character in black on a black

background.

3.
```
charNormal$:
strb char,[bufferStop]
ldr r0,=terminalColour
ldrb r0,[r0]
strb r0,[bufferStop,#1]
add bufferStop,#2
add x,#1
```

Store the current character in the string and the terminalColour to the end of the terminalBuffer and then increment it and x.

4.
```
charLoopContinue$:
cmp x,#128
blt noScroll$
```

Check if x is at the end of a line; 128.

5.
```
mov x,#0
subs r0,bufferStop,view
addlt r0,#128*128*2
cmp r0,#128*(768/16)*2
```

Set x back to 0 and check if we're currently showing more than one screen. Remember, we're using a circular buffer, so if the difference between bufferStop and view is negative, we're actually wrapping around the buffer.

6.
```
addge view,#128*2
```

Add one lines worth of bytes to the view address.

7.
```
teq view,taddr
subeq
view,taddr,#128*128*2
```

If the view address is at the end of the buffer we subtract the buffer length from it to move it back to the start. I set taddr to the address of the end of the buffer at the beginning.

8.
```
noScroll$:
teq bufferStop,taddr
subeq
bufferStop,taddr,#128*128*2
```

If the stop address is at the end of the buffer we subtract the buffer length from it to move it back to the start. I set taddr to the address of the end of the buffer at the beginning.

9.
```
teq bufferStop,bufferStart
addeq bufferStart,#128*2
```

Check if bufferStop equals bufferStart. If so, add one line to bufferStart.

10.
```
teq bufferStart,taddr
subeq
bufferStart,taddr,#128*128*2
```

If the start address is at the end of the buffer we subtract the buffer length from it to move it back to the start. I set taddr to the address of the end of the buffer at the beginning.

```
subs length,#1
add string,#1
bgt charLoop$
```

Loop until the string is done.

5.
```
charLoopBreak$:
sub taddr,#128*128*2
sub taddr,#terminalBuffer-
terminalStart
str bufferStop,[taddr,#terminalStop-
terminalStart]
str view,[taddr,#terminalView-
terminalStart]
str bufferStart,[taddr]

pop {r4,r5,r6,r7,r8,r9,r10,r11,pc}
.unreq bufferStart
.unreq taddr
.unreq x
.unreq string
.unreq length
.unreq char
.unreq bufferStop
.unreq view
```

Store back the variables and return.

This method allows us to print arbitrary text to the screen. Throughout, I've been using the colour variable, but no where have we actually set it. Normally, terminals use

special combinations of characters to change the colour. For example ASCII Escape ($1b_{16}$) followed by a number 0 to f in hexadecimal could set the foreground colour to that CGA colour number. You can try implementing this yourself; my version is in the further examples section on the download page.

4 Standard Input

> By convention, in many programming languages, every program has access to stdin and stdout, which are an input and and output stream linked to the terminal. This is still true on graphical programs, though many don't use it.

Now we have an output terminal that in theory can print out text and display it. That is only half the story however, we want input. We want to implement a method, ReadLine, which stores the next line of text a user types to a location given in r0, up to a maximum length given in r1, and returns the length of the string read in r0. The tricky thing is, the user annoyingly wants to see what they're typing as they type it, they want to use backspace to delete mistakes and they want to use return to submit commands. They probably even want a flashing underscore character to indicate the computer would like input! These perfectly reasonable requests make this method a real challenge. One way to achieve all of this is to store the text they type in memory somewhere along with its length, and then after every character, move the terminalStop address back to where it started when ReadLine was called and calling Print. This means we only have to be able to manipulate

a string in memory, and then make use of our Print function.

Lets have a look at what ReadLine will do:

1. If the maximum length is 0, return 0
2. Retrieve the current values of terminalStop and terminalView
3. If the maximum length is bigger than half the buffer size, set it to half the buffer size
4. Subtract one from maximum length to ensure it can store our flashing underscore or a null terminator
5. Write an underscore to the string
6. Write the stored terminalView and terminalStop addresses back to the memory
7. Call Print on the current string
8. Call TerminalDisplay
9. Call KeyboardUpdate
10. Call KeyboardGetChar
11. If it is a new line character go to 16
12. If it is a backspace character, subtract 1 from the length of the string (if it is > 0)
13. If it is an ordinary character, write it to the string (if the length < maximum length)
14. If the string ends in an underscore, write a space, otherwise write an underscore
15. Go to 6
16. Write a new line character to the end of the string
17. Call Print and TerminalDisplay
18. Replace the new line with a null terminator
19. Return the length of the string

Convince yourself that this will work, and then try to implement it yourself. My implementation is given below:

1.
```
.globl ReadLine
ReadLine:
teq r1,#0
moveq r0,#0
moveq pc,lr
```

   Quick special handling for the zero case, which is otherwise difficult.

2.
```
string .req r4
maxLength .req r5
input .req r6
taddr .req r7
length .req r8
view .req r9

push {r4,r5,r6,r7,r8,r9,lr}

mov string,r0
mov maxLength,r1
ldr taddr,=terminalStart
ldr input,[taddr,#terminalStop-
terminalStart]
ldr view,[taddr,#terminalView-
terminalStart]
mov length,#0
```

   As per the general theme, I do a lot of initialisations early. input contains the value of terminalStop and view contains terminalView. Length starts at 0.

3.
```
cmp maxLength,#128*64
```

```
movhi maxLength,#128*64
```

We have to check for unusually large reads, as we can't process them beyond the size of the terminalBuffer (I suppose we CAN, but it would be very buggy, as terminalStart could move past the stored terminalStop).

4.
```
sub maxLength,#1
```

Since the user wants a flashing cursor, and we ideally want to put a null terminator on this string, we need 1 spare character.

5.
```
mov r0,#'_'
strb r0,[string,length]
```

Write out the underscore to let the user know they can input.

6.
```
readLoop$:
str input,[taddr,#terminalStop-terminalStart]
str view,[taddr,#terminalView-terminalStart]
```

Save the stored terminalStop and terminalView. This is important to reset the terminal after each call to Print, which changes these variables. Strictly speaking it can change terminalStart too, but this is irreversible.

7.
```
mov r0,string
```

```
mov r1,length
add r1,#1
bl Print
```

Write the current input. We add 1 to the length for the underscore.

8.
```
bl TerminalDisplay
```

Copy the new text to the screen.

9.
```
bl KeyboardUpdate
```

Fetch the latest keyboard input.

10.
```
bl KeyboardGetChar
```

Retrieve the key pressed.

11.
```
teq r0,#'\n'
beq readLoopBreak$
teq r0,#0
beq cursor$
teq r0,#'\b'
bne standard$
```

Break out of the loop if we have an enter key. Also skip these conditions if we have a null terminator and process a backspace if we have one.

12.
```
delete$:
cmp length,#0
subgt length,#1
b cursor$
```

Remove one from the length to delete a character.

13.
```
standard$:
cmp length,maxLength
bge cursor$
strb r0,[string,length]
add length,#1
```

Write out an ordinary character where possible.

14.
```
cursor$:
ldrb r0,[string,length]
teq r0,#'_'
moveq r0,#' '
movne r0,#'_'
strb r0,[string,length]
```

Load in the last character, and change it to an underscore if it isn't one, and a space if it is.

15.
```
b readLoop$
readLoopBreak$:
```

Loop until the user presses enter.

16.
```
mov r0,#'\n'
strb r0,[string,length]
```

Store a new line at the end of the string.

17.
```
str input,[taddr,#terminalStop-
terminalStart]
```

```
str view,[taddr,#terminalView-
terminalStart]
mov r0,string
mov r1,length
add r1,#1
bl Print
bl TerminalDisplay
```

Reset the terminalView and terminalStop and then Print and TerminalDisplay the final input.

18.
```
mov r0,#0
strb r0,[string,length]
```

Write out the null terminator.

19.
```
mov r0,length
pop {r4,r5,r6,r7,r8,r9,pc}
.unreq string
.unreq maxLength
.unreq input
.unreq taddr
.unreq length
.unreq view
```

Return the length.

5 The Terminal: Rise of the Machine

So, now we can theoretically interact with the user on the terminal. The most obvious thing to do is to put this to the test! In 'main.s' delete everything after **bl UsbInitialise** and copy in the following code:

```
```

```
reset$:
	mov sp,#0x8000
	bl TerminalClear

	ldr r0,=welcome
	mov r1,#welcomeEnd-welcome
	bl Print
loop$:
	ldr r0,=prompt
	mov r1,#promptEnd-prompt
	bl Print

	ldr r0,=command
	mov r1,#commandEnd-command
	bl ReadLine

	teq r0,#0
	beq loopContinue$

	mov r4,r0

	ldr r5,=command
	ldr r6,=commandTable

	ldr r7,[r6,#0]
	ldr r9,[r6,#4]
	commandLoop$:
		ldr r8,[r6,#8]
		sub r1,r8,r7

		cmp r1,r4
		bgt commandLoopContinue$

		mov r0,#0
		commandName$:
			ldrb r2,[r5,r0]
			ldrb r3,[r7,r0]
			teq r2,r3
			bne commandLoopContinue
			$
```

```
                add r0,#1
                teq r0,r1
                bne commandName$

            ldrb r2,[r5,r0]
            teq r2,#0
            teqne r2,#' '
            bne commandLoopContinue$

            mov r0,r5
            mov r1,r4
            mov lr,pc
            mov pc,r9
            b loopContinue$

        commandLoopContinue$:
            add r6,#8
            mov r7,r8
            ldr r9,[r6,#4]
            teq r9,#0
            bne commandLoop$

        ldr r0,=commandUnknown
        mov r1,#commandUnknownEnd-
        commandUnknown
        ldr r2,=formatBuffer
        ldr r3,=command
        bl FormatString

        mov r1,r0
        ldr r0,=formatBuffer
        bl Print

loopContinue$:
        bl TerminalDisplay
        b loop$

echo:
        cmp r1,#5
        movle pc,lr
```

```
        add r0,#5
        sub r1,#5
        b Print

ok:
        teq r1,#5
        beq okOn$
        teq r1,#6
        beq okOff$
        mov pc,lr

        okOn$:
                ldrb r2,[r0,#3]
                teq r2,#'o'
                ldreqb r2,[r0,#4]
                teqeq r2,#'n'
                movne pc,lr
                mov r1,#0
                b okAct$

        okOff$:
                ldrb r2,[r0,#3]
                teq r2,#'o'
                ldreqb r2,[r0,#4]
                teqeq r2,#'f'
                ldreqb r2,[r0,#5]
                teqeq r2,#'f'
                movne pc,lr
                mov r1,#1

        okAct$:
                mov r0,#16
                b SetGpio

.section .data
.align 2
welcome: .ascii "Welcome to Alex's OS -
Everyone's favourite OS"
welcomeEnd:
```

```
.align 2
prompt: .ascii "\n> "
promptEnd:
.align 2
command:
	.rept 128
		.byte 0
	.endr
commandEnd:
.byte 0
.align 2
commandUnknown: .ascii "Command `%s' was not recognised.\n"
commandUnknownEnd:
.align 2
formatBuffer:
	.rept 256
		.byte 0
	.endr
formatEnd:

.align 2
commandStringEcho: .ascii "echo"
commandStringReset: .ascii "reset"
commandStringOk: .ascii "ok"
commandStringCls: .ascii "cls"
commandStringEnd:

.align 2
commandTable:
.int commandStringEcho, echo
.int commandStringReset, reset$
.int commandStringOk, ok
.int commandStringCls, TerminalClear
.int commandStringEnd, 0
```

This code brings everything together into a simple command line operating system. The commands available are echo, reset, ok and cls. echo copies any text after it back to the

terminal, reset resets the operating system if things go wrong, ok has two functions: ok on turns the OK LED on, and ok off turns the OK LED off, and cls clears the terminal using TerminalClear.

Have a go with this code on the Raspberry Pi. If it doesn't work, please see our troubleshooting page.

When it works, congratulations you've completed a basic terminal Operating System, and have completed the input series. Unfortunately, this is as far as these tutorials go at the moment, but I hope to make more in the future. Please send feedback to awc32@cam.ac.uk.

You're now in position to start building some simple terminal Operating Systems. My code above builds up a table of available commands in commandTable. Each entry in the table is an int for the address of the string, and an int for the address of the code to run. The last entry has to be commandStringEnd, 0. Try implementing some of your own commands, using our existing functions, or making new ones. The parameters for the functions to run are r0 is the address of the command the user typed, and r1 is the length. You can use this to pass inputs to your commands. Maybe you could make a calculator program, perhaps a drawing program or a chess program. Whatever ideas you've got, give them a go!

Spot a mistake? You can help improve this tutorial on GitHub.

Based on contributions at https://
github.com/chadderz121/bakingpi-www.

Downloads

This page contains download links for the GNU toolchains, as well as the model answers to each lesson.

1 GNU Toolchain

1.1 Microsoft Windows

For Microsoft Windows, I use the YAGARTO and MinGW packages.

Please visit the YAGARTO website and download and install YAGARTO Tools and YAGARTO GNU ARM toolchain for Windows. MinGW can be downloaded from here. You may need to restart your computer for this to work (honestly).

Note: YAGARTO must be installed to a path with no spaces, e.g. 'C:\YAGARTO\' not 'C:\Program Files\YAGARTO\'.

1.2 Mac OS X

For Mac OS X, I use the YAGARTO packages.

Please visit the YAGARTO website and download and install YAGARTO GNU ARM toolchain for Mac OS X.

1.3 Linux

There are a number of options for getting the GNU ARM toolchain on Linux.

1.3.1 Prebuilt

You can download a prebuilt toolchain using
the following commands:

```
$ wget http://www.cl.cam.ac.uk/freshers/raspberrypi/tuto
--2012-08-16 18:26:29--  http://www.cl.cam.ac.uk/fresher
Resolving www.cl.cam.ac.uk (www.cl.cam.ac.uk)... 128.232
Connecting to www.cl.cam.ac.uk (www.cl.cam.ac.uk)|128.23
HTTP request sent, awaiting response... 200 OK
Length: 32108070 (31M) [application/x-bzip2]
Saving to: `arm-none-eabi.tar.bz2'

100%[======================================>] 32,108,070

2012-08-16 18:27:39 (467 KB/s) - `arm-none-eabi.tar.bz2'

$ tar xjvf arm-none-eabi.tar.bz2
arm-2008q3/arm-none-eabi/
arm-2008q3/arm-none-eabi/lib/
arm-2008q3/arm-none-eabi/lib/libsupc++.a
arm-2008q3/arm-none-eabi/lib/libcs3arm.a
...
arm-2008q3/share/doc/arm-arm-none-eabi/info/gprof.info
arm-2008q3/share/doc/arm-arm-none-eabi/info/cppinternals
arm-2008q3/share/doc/arm-arm-none-eabi/LICENSE.txt

$ export PATH=$PATH:$HOME/arm-2008q3/bin
```

1.3.2 apt-get

Some Linux distributions including Ubunutu
offer the ARM GNU Toolchain via apt-get.
Run the following command:

```
$ sudo apt-get install gcc-arm-none-eabi
```

1.3.3 Build from source

Linux users may wish to build their own

cross-compiler toolchain. This will require downloading and building the binutils and gcc packages from GNU. The binutils package contains the basic tools for building executables, including the assembler, the linker, a disassembler, and tools to manipulate object and binary files. These two packages are built separately but should be installed into the same destination directory. Make an area in your home directory to build your development kit.

```
$ cd
$ mkdir devkit
$ mkdir devkit-build
$ cd devkit-build
```

Download the binutils-X.XX.tar.bz2 package from GNU binutils into the devkit-build directory and uncompress it with the *tar jxv binutils-X.XX.tar.bz2* command. Replace the X.XX with the current version number, such as 2.24 or 2.25 which have both been used successfully for this course. A note about make commands: if your development system has multiple processors or cores, you can use then to build and compile in parallel by adding *-j #cores* A note about the *--program-prefix* option: the trailing dash (-) is necessary to make sure the command names match helper templates used later in the course. Build the tools with these steps, replacing X.XX with your specific binutils version number:

```
$ cd $HOME/devkit-build
$ mkdir binutils-build
$ cd binutils-build
$ ../binutils-X.XX/configure --prefix=$HOME/devkit/ \
--program-prefix=arm-none-eabi- --target=arm-none-eabi -
```

```
$ make
$ make check
$ make install
$ cd ..
```

The gcc package contains a C compiler. Download the gcc-X.XX.tar.bz2 package from GCC, The GNU Compiler Collection into the devkit-build directory and uncompress it with the *tar jxv gcc-X.XX.tar.bz2* command. Replace the X.XX with the current version number, such as 4.8.2 or 5.1 which have both been used successfully for this course. Build the tools with these steps, replacing X.XX with your specific gcc version number, and add the -j #cores option, if desired

```
$ cd $HOME/devkit-build
$ mkdir gcc-build
$ cd gcc-build
$ ../gcc-X.XX/configure --prefix=$HOME/devkit/ \
--program-prefix=arm-none-eabi- --target=arm-none-eabi -
--without-headers --with-newlib --with-as=$HOME/devkit/b
--with-ld=$HOME/devkit/bin/arm-none-eabi-ld --enable-lar
$ make all-gcc
$ make check all-gcc
$ make install-gcc
$ make all-target-libgcc
$ make check all-target-libgcc
$ make install-target-libgcc
$ cd ..
```

Now that you have a custom cross-compiler in your $HOME/devkit directory, you will need to add this directory to your shell's PATH environment variable to be able to run the tools later in the course. Each time you are ready to use the toolchain, run the following shell command:

```
$ export PATH=$PATH:$HOME/devkit/bin
```

2 OS Template

The OS Template file is one I have created which contains enough instructions for the compiler to create a basic Operating System for the Raspberry Pi. It contains no actual assembly code, just a **Makefile script** and a **Linker script**.

Download Template.

Download Template for USB Operating System.

3 Lesson Solutions

3.1 Lesson 1: OK01

Full Solution

3.2 Lesson 2: OK02

Full Solution

3.3 Lesson 3: OK03

Full Solution

Extension Solution

3.4 Lesson 4: OK04

Full Solution

3.5 Lesson 5: OK05

Full Solution

3.6 Lesson 6: Screen01

Full Solution

3.7 Lesson 7: Screen02

Full Solution

3.8 Lesson 8: Screen03

Full Solution

3.9 Lesson 9: Screen04

Full Solution

3.10 Lesson 10: Input01

Lesson Template

Full Solution

3.11 Lesson 11: Input02

Full Solution

4 Example Operating Systems

Here are some example operating systems for you to learn from. If you've coded an operating system that you think others could benefit from, please email me at awc32@cam.ac.uk.

Name	Author	Description
Coloured CLI	Alex Chadwick	This example is a small extension to Input01, featuring a

		coloured text terminal, rather than a black and white one. Special characters are used to change the colour.
Pascal OSs	Marten van der Honing	A few small OSs written is Pascal based on this course, and beyond.

Table 4.1 Example Operating Systems

5 Fonts

Below are some fonts for you to use in your Operating Systems.

5.1 Monospace, Monochrome 8x16

These fonts are the simplest ones available. They use a 1 to represent a white pixel, a 0 to represent a black pixel, and having representations for the first 128 ASCII characters. The use 16 bytes per character, arranged such that each byte is one complete row, going **right** to **left** with higher bits, going top to bottom with later bytes.

The tutorial used to suggest these fonts were stored in the opposite direction along the rows. The lowest bit is the rightmost pixel, the highest bit is the leftmost.

Font	Image	License
Monospace Default		Free to use/ redistribute

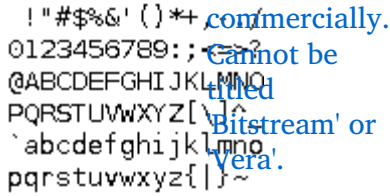
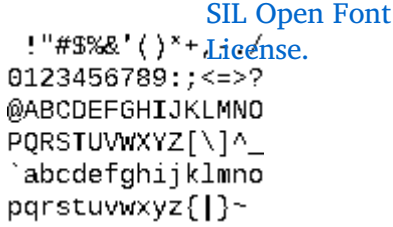
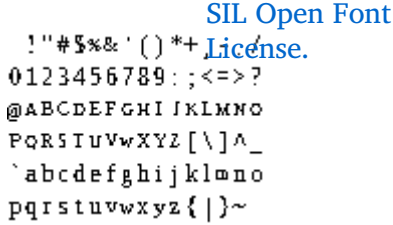
		commercially. Cannot be titled 'Bitstream' or 'Vera'.
Liberation Mono		SIL Open Font License.
Liberation Serif Mono		SIL Open Font License.

Table 5.1.1 Monospace, Monochrome 8x16 Fonts

6 USB driver (CSUD) Source

The source code for CSUD (Chadderz's Simple USB driver) used in the tutorials is available here: https://github.com/Chadderz121/csud.

Spot a mistake? You can help improve this tutorial on GitHub.

Troubleshooting

This course has not yet been updated to work with the Raspberry Pi models B+ and A+. Some elements may not work, in particular the first few lessons about the LED. It has also not been updated for Raspberry Pi v2.

There is little more satisfying than when an Operating System you've written works perfectly, however this is, unfortunately, rarely the case. I lost count long ago how many Operating Systems I've written that didn't work in making this course. This page contains advice for what to do when things just aren't working. It is broken down into compile errors that happen before you can even make the Operating System, load errors that prevent your Operating System doing anything, and runtime errors, where your Operating System doesn't do the correct thing. I've also added specific help on each of the tutorial Operating Systems, such as things that can commonly go wrong. If you have a problem that isn't explained here, and think others my be able to benefit from your experience, send an email to me at awc32@cam.ac.uk, and I will add it here. It is also well worth taking a look at the Raspberry Pi forums to see if anyone else has run into the same problem.

1 Compiler Errors

Compiler errors are errors that occur when

the **make** command runs on your Operating System. I've also included some common warnings too.

1.1 Error: bad instruction

```
source/file.s:8: Error: bad instruction `sdd r0,r1'
```

This error occurs when you use a command that doesn't exist. First of all, check that you haven't mistyped the command. If you're using condition codes as well as other options such as an **str** command with a **b** suffix to only store a byte, and a **eq** suffix to only store if the last condition was equal, the correct order is in fact **streqb** not **strbeq**.

1.2 Error: immediate expression requires a # prefix

```
source/file.s:32: Error: immediate expression requires a
```

This means that you're trying to use a constant number, such as adding the number one, but forgot to put a # (e.g. **add r3,4** should be **add r3,#4**. You must do so wherever you use a constant on a command that normally uses registers. This is true even for calculated constants such as **#3*4**.

1.3 Error: ARM register expected

```
source/file.s:24: Error: ARM register expected -- `add 0
```

This means that you typed something that was not a register, when a register was expected. Double check your spelling, especially if you're using **.req**. If you are, make sure you haven't used **.unreq** between the **.req** and this command.

1.4 Error: unknown pseudo-op

```
source/file.s: Error: unknown pseudo-op: `.suction'
```

This error occurs when you use a pseudo operation that doesn't exit. Check your spelling.

1.5 Error: invalid constant (number) after fixup

```
source/file.s:24: Error: invalid constant (c21) after fi
```

This error occurs when you use a constant which does not meet the requirements of the function. The most common example of this is the **mov** instruction, which only allows numbers which can be represented as an 8 bit number, shifted left by an even number. For example $c21_{16} = 110000100001_2$ and so cannot be represented in a **mov**, but $c20_{16} = 110000100000_2 = 11000010_2 << 4$, and so is valid in a **mov**. Much the same rules apply to most constants in functions. Remember, to load in any constant, use **ldr r0, =value**.

1.6 warning : end of file not at end of a line; newline inserted

```
source/file.s: warning : end of file not at end of a lin
```

This means the last line in your file is not empty. You can ignore this, but to fix it just add a new line at the end.

1.7 undefined reference

```
.text(+0x18): undefined reference to `label'
```

This means you've used a label which the linker can't find. This is probably due to a misspelling. Remember that labels are case sensitive and that labels in different files require **.globl** commands before they're accessible in other files.

1.8 `section' referenced in section `section' of build/file.o: defined in discarded section `section' of build/file.o

```
`.trxt' referenced in section `.init' of build/main.o: c
```

This means that you've used a **.section** command, but you've specified a section other than .init, .text or .data. Only these sections are copied into the kernel.img file, any others are discarded, hence the error is saying that some of your code was discarded. Check your spelling on **.section** commands.

1.9 arm-none-eabi-ld: no input files

```
arm-none-eabi-ld: no input files
```

This error means that the linker hasn't found your code. Double check you've got a source directory with .s files within it like main.s. Make sure you haven't got something like main.s.txt.

1.10 make: *** No targets specified and no makefile found. Stop.

```
make: *** No targets specified and no makefile found.  S
```

This error is caused by running make in the wrong directory. The command line must

have the same working directory as the file makefile which is in the template. Use the 'cd' command to change directories, then run make again.

1.11 Windows Only: make: Interrupt/Exception caught (code = 0xc00000fd, addr = 0x425073)

```
make: Interrupt/Exception caught (code = 0xc00000fd, add
```

This is an error that can occur on Windows when YARGTO has been installed in a directory with a space in it's name, for example: C:\Program Files (x86)\YAGARTO \. To fix, please reinstall YAGARTO in a directory with no spaces such as C: \YAGARTO\.

1.12 Linux 64 bit Only: arm-none-eabi-as: No such file or directory

```
bash: arm-none-eabi-as: No such file or directory
```

This error is caused by running the Linux version of the toolchain on a 64 bit machine without 32 bit compatibility libraries. These can be retrieved easily using:

```
sudo apt-get install ia32-libs
```

2 Load Errors

Load errors are errors that occur that prevent your Operating System from giving any output. This can be the hardest to diagnose and fix. Unfortunately, by their nature, they give off no indication of what is wrong.

The first thing you should check is that the tutorial answer does work. This confirms that you're installing things correctly, that your Raspberry Pi is not physically damaged and that your SD card works. If the answer doesn't work, make sure Linux still does. If it doesn't you may have a problem with your SD card or Raspberry Pi physically. Reimage the SD card or get a new one. If Linux does work but the tutorial does not, you may not be installing the Operating System correctly. Double check you're replacing kernel.img in the FAT partition of the SD card.

If the answer does work but your attempt does not, then we know it is something in your code. On the later tutorials, try altering the start of your code to turn on the OK LED, just so you know if it boots at all. If not, double check that you have some code in the .init section which branches into your .text section. Try enabling the OK LED from the .init section.

Ultimately what we need is some output. If you can get the LED to turn on from your early code, then this is just a runtime error. If placing that code in the .init section still doesn't enable output, it may be worth going back to the template and copying in your code bit by bit until it stops working. Sometimes you never do find the error; In the past I've ended up copying the entire code back into the template and suddenly it worked.

3 Runtime Errors

Behind load errors, runtime errors are the hardest to diagnose and fix. These occur

when your Operating System just doesn't do what you want.

The most important thing is to get information out of the system. The OK LED is very useful for this. If the Operating System seems to stop, or get stuck, try turning on the LED just before and just after various commands. If it turns on when just before an instruction, but won't on the instruction afterwards, then we know this is the problem. Please remember that turning on the LED will generally alter r0 to r3, so use **push {r0,r1,r2,r3}** and **pop {r0,r1,r2,r3}** to preserve these registers. If you're in looping code, try flashing the LED in the loop to see how many loops actually happen.

If you're in the later tutorials make sure to use the screen for information. Write out text about the current status, values, etc, in order to learn what is going on. Once you've got some idea, have a look at the following common problems to see if you can spot what has happened.

Remember, think outside the box. Perhaps a function you wrote ages ago had a bug you never noticed.

3.1 Alignment

One of the most subtle runtime errors is the ARM alignment constraint. Any **str** or **ldr** instruction will not function correctly unless the computed address is a multiple of the size of the data being read. For example, if you're using **ldr r0, [r1,#2]** , then the value of **r1,#2** must be a multiple of 4. If not unpredictable results occur. You should

always be able to guarantee this is this case. If you're referring to a label, make sure you have a **.align 2** command BEFORE the label. This will ensure that the label's address is a multiple of $2^2 = 4$. You can use **.align 3** to align to a multiple of $2^3 = 8$, etc.

3.2 Hanging

A processor 'hangs' (stops) if it encounters a bad instruction or a bad address. If your code gets stuck on a branch, load or store command, this is likely to be the problem. You can use a condition around turning on the OK LED to check this.

3.3 Infinite Loops

Similarly to hanging, a processor can easily get stuck in loops. If the processor reaches one of your loops, but never leaves, this could well be the problem. Double check the conditions for leaving the loop will be satisfied.

4 General Advice

To help you, every time you compile a kernel, two extra files are compiled. kernel.list contains a direct listing of all the assembly code, and kernel.map contains a map of all your labels. You can use these files to mentally simulate the processor and check it will do the correct thing. The processor starts at address 0 with all registers in an undefined state. Try mentally checking that the processor will do what you want. For things like alignment issues, you can double check everything is as it should be with

kernel.map.

5 Tutorial Specific Advice

5.1 OK05 Doesn't Flash; Light Stays On

If you followed a previous version of this tutorial, a common problem on OK05 is to have the OK LED stay on continuously rather than flashing a pattern. This is caused by a change in the way the modern bootloaders load the kernel; they load it at address 0x8000 not 0. Either replace makefile and kernel.ld with the ones currently in the template. Or alternatively add the line **kernel_old=1** to the file config.txt on the SD card, or create the file with this line in it.

5.2 Screen01 Displays Nothing

If you followed a previous version of this tutorial, this was a common problem. The code in framebuffer.s has been altered to fix this problem. Specifically, it is necessary to add 0x40000000 to the address of FrameBufferInfo before writing it to the mailbox.

5.3 Screen02 Displays Nothing

See OK05 Doesn't Flash; Light Stays On.

5.4 Input01 Displays Nothing

First of all, check if this is an issue with the screen or the keyboard by running the solution to Input02. It prints a message to the screen before receiving keyboard input. If

nothing shows on Input02, then see the help for Screen01. If it does display, but you still can't type, then your keyboard may be incompatible with my USB driver. Unfortunately, due to it's basic code, the driver doesn't support every keyboard. Try to find other USB keyboards to use. I've personally tested 11 brands of keyboard, of which 6 worked.

Spot a mistake? You can help improve this tutorial on GitHub.

Based on contributions at https://github.com/chadderz121/bakingpi-www.

ARM Reference

This page contains a reference for the ARMv6 instruction set, register set, and also the GNU Assembler program syntax.

1 Tutorial ARM Instructions

The following is a list of all the instruction boxes in the courses in order.

ldr reg,=val puts the number **val** into the register named **reg**.

mov reg,#val puts the number **val** into the register named **reg**.

lsl reg,#val shifts the binary representation of the number in **reg** by **val** places to the left.

str reg,[dest,#val] stores the number in **reg** at the address given by **dest** + **val**.

name: labels the next line **name**.

b label causes the next line to be executed to be **label**.

sub reg,#val subtracts the number **val** from the value in **reg**.

cmp reg,#val compares the value in **reg** with the number **val**.

Suffix **ne** causes the command to be executed only if the last comparison determined that the numbers were not equal.

.globl lbl makes the label **lbl** accessible from other files.

mov reg1,reg2 copies the value in **reg2** into **reg1**.

Suffix **ls** causes the command to be executed only if the last comparison determined that the first number was less than or the same as the second. Unsigned.

Suffix **hi** causes the command to be executed only if the last comparison determined that the first number was higher than the second. Unsigned.

push {reg1,reg2,...} copies the registers in the list **reg1,reg2,...** onto the top of the stack. Only general purpose registers and lr can be pushed.

bl lbl sets **lr** to the address of the next instruction and then branches to the label **lbl**.

add reg,#val adds the number **val** to the contents of the register **reg**.

Argument shift **reg,lsl #val** shifts the binary representation of the number in **reg** left by **val** before using it in the operation before.

lsl reg,amt shifts the binary representation of the number in **reg** left by the number in **amt**.

str reg, [dst] is the same as **str reg, [dst,#0]** .

pop {reg1,reg2,...} copies the values from the top of the stack into the register

list **reg1,reg2,....** Only general purpose registers and pc can be popped.

alias .req reg sets **alias** to mean the register **reg**.

.unreq alias removes the alias **alias**.

lsr dst,src,#val shifts the binary representation of the number in **src** right by **val**, but stores the result in **dst**.

and reg,#val computes the Boolean and function of the number in **reg** with **val**.

teq reg,#val checks if the number in **reg** is equal to **val**.

ldrd regLow,regHigh, [src,#val] loads 8 bytes from the address given by the number in **src** plus **val** into **regLow** and **regHigh**.

.align num ensures the address of the next line is a multiple of $2^{\mathbf{num}}$.

.int val outputs the number **val**.

tst reg,#val computes **and reg,#val** and compares the result with 0.

mla dst,reg1,reg2,reg3 multiplies the values from **reg1** and **reg2**, adds the value from **reg3** and places the least significant 32 bits of the result in **dst**.

strh reg, [dest] stores the low half word number in **reg** at the address given by **dest**.

Spot a mistake? You can help improve this tutorial on GitHub.